Overview

We have multiple microservices and endpoints in our project which have different load handling capacity based on the types of operations they perform. For example, the login endpoint associated with the registry has greater load handling capacity as compared to the plotting microservice since plotting is both a memory and time intensive operation because of the various operations it performs. As such, we are expecting a greater load handling capacity from the login, signup, endpoints in contrast to the plotting and history service endpoints, for which we are expecting a lower load handling capacity. We haven't quantified the estimated values but we'll get a good idea about it after running multiple tests on Apache Jmeter. We have set an error limit of 10% which we are quantifying as an acceptable error rate for deployment purpose. We'll explore the different microservices in detail and will also observe their performances when they are used alone versus when they are combined with the gateway. Later on, we'll discuss ways in which we can improve on our current system to handle even more number of simultaneous requests effectively.

These are a few terms to denote how we are hitting that specific endpoint. These terms will be used in the headers for quick understanding of the way the endpoint is reached:

1) Direct – Sending multiple requests to that specific microservice directly

2) Gateway – Sending multiple requests to the endpoint through gateway

3) Kubernetes - Sending multiple requests to the endpoint while using Kubernetes

Types of Graphs

1) Graph Results – Shows the average, media, deviation, etc of the number of samples over time

2) Active Threads – Displays the number of active threads over time

3) Bytes Throughput Per Second – Shows the bytes received and bytes sent per second

4) Response Codes Per Second – A graph of success and failed responses across time

5) Response Latencies Over Time – Shows the latency over time

6) Response Times Distribution – Displays the number of responses per ms

1) Login - Direct

Microservice – Dev Registry

Language - NodeJS

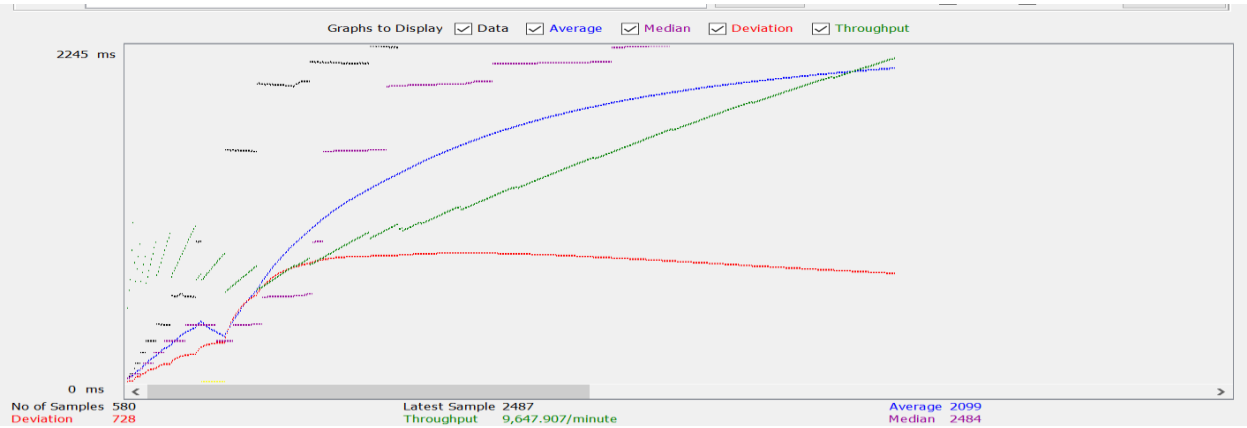
URL - /registry/api/v1/user/login

Max Number of users: 580

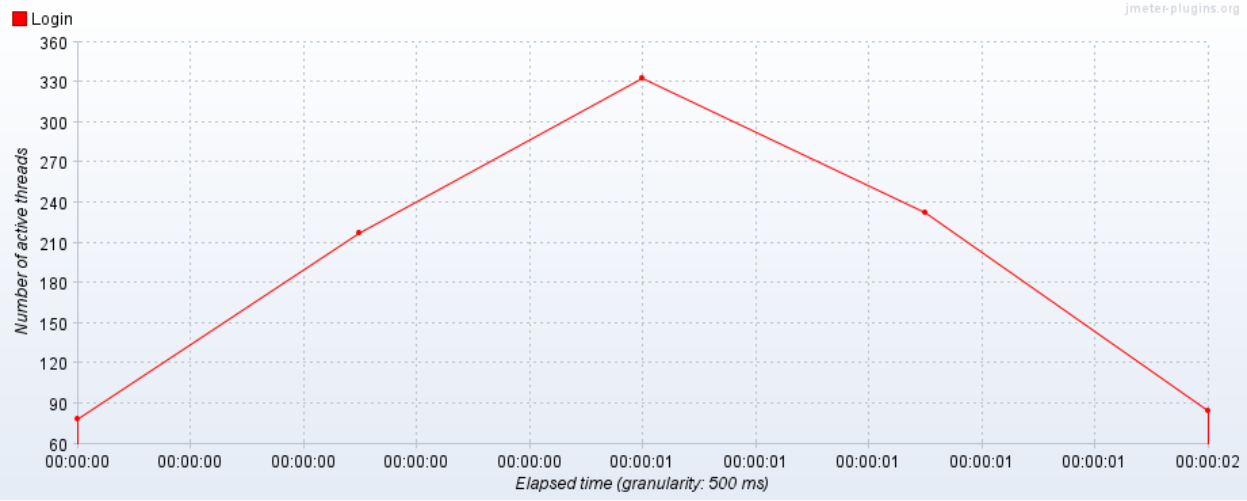
Error rate: 9%

Throughput: 263.3/sec

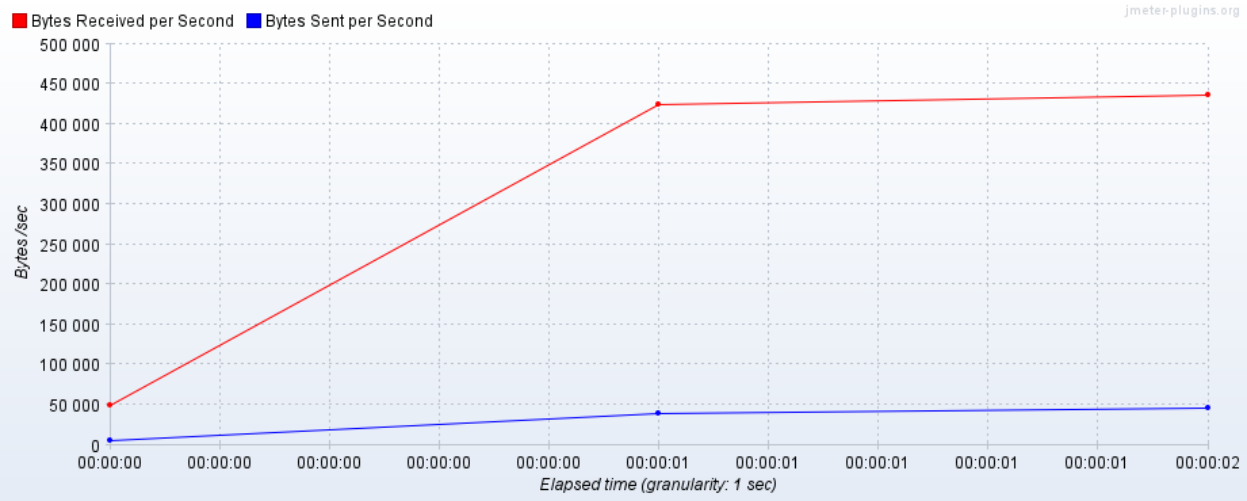
Graph Results



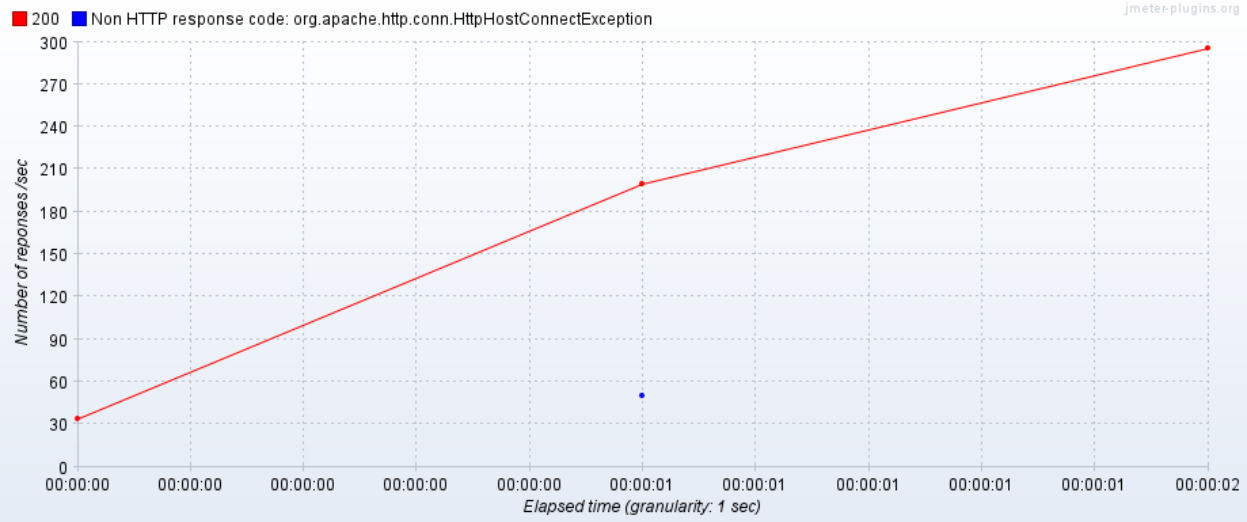
Active Threads



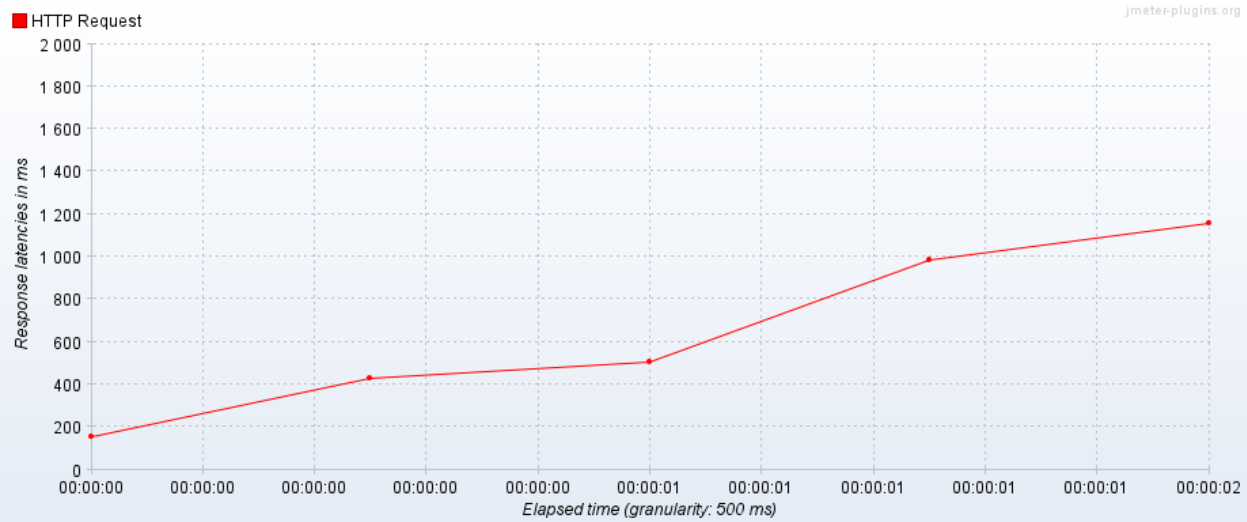
Bytes Throughput



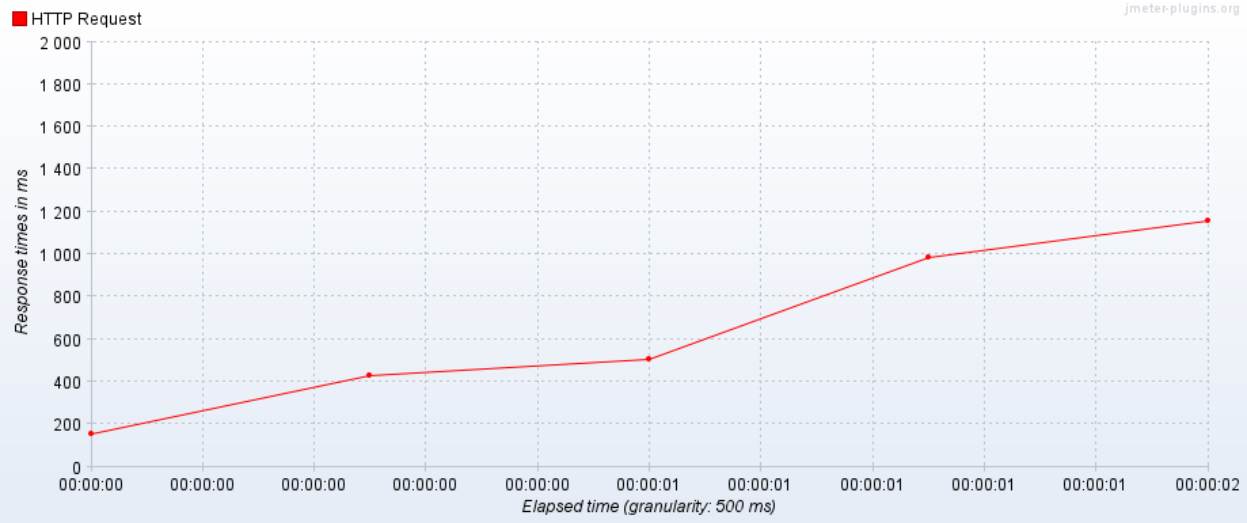
Response Codes Per Second



Response Latencies Over Time



Response Times Distribution



2) Login – Gateway

Microservice – Dev Registry

Language – Spring Boot, NodeJS

URL - /login

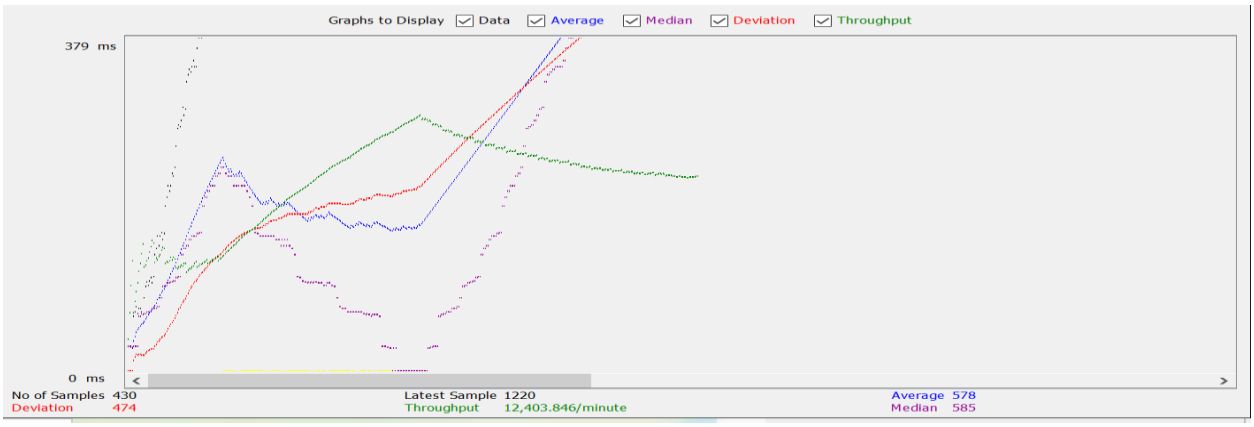
Max Number of users: 430

Error rate: 11%

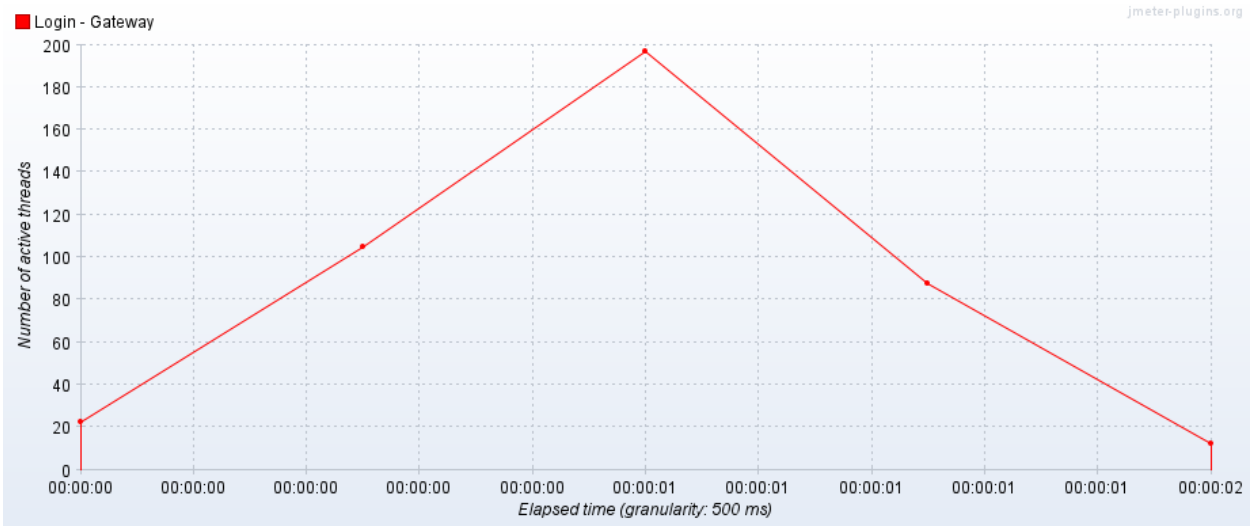
Throughput: 254.6/sec

Acceptable users: 430

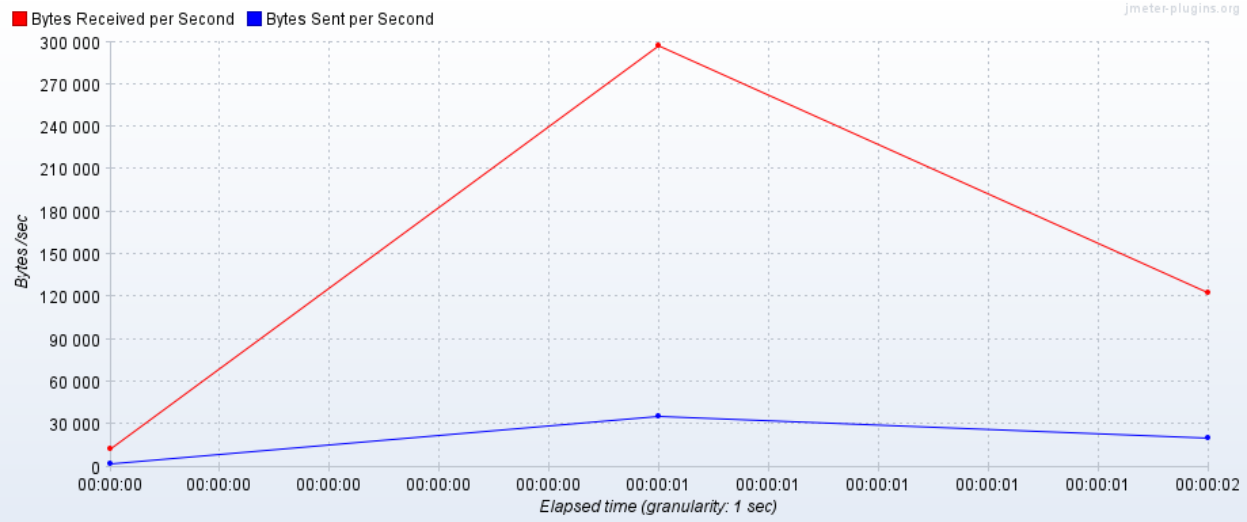
Graph Results



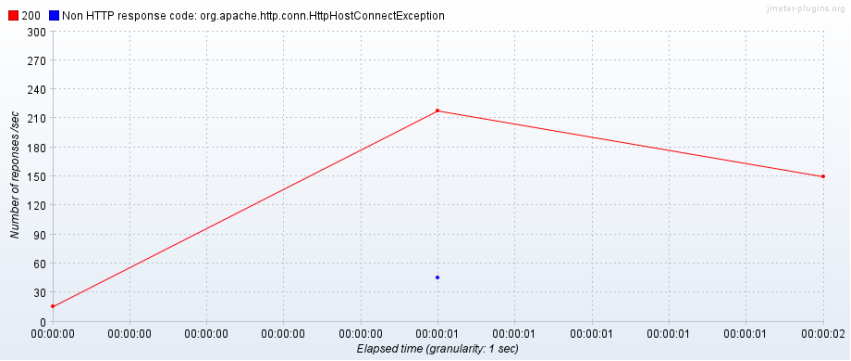
Active Threads



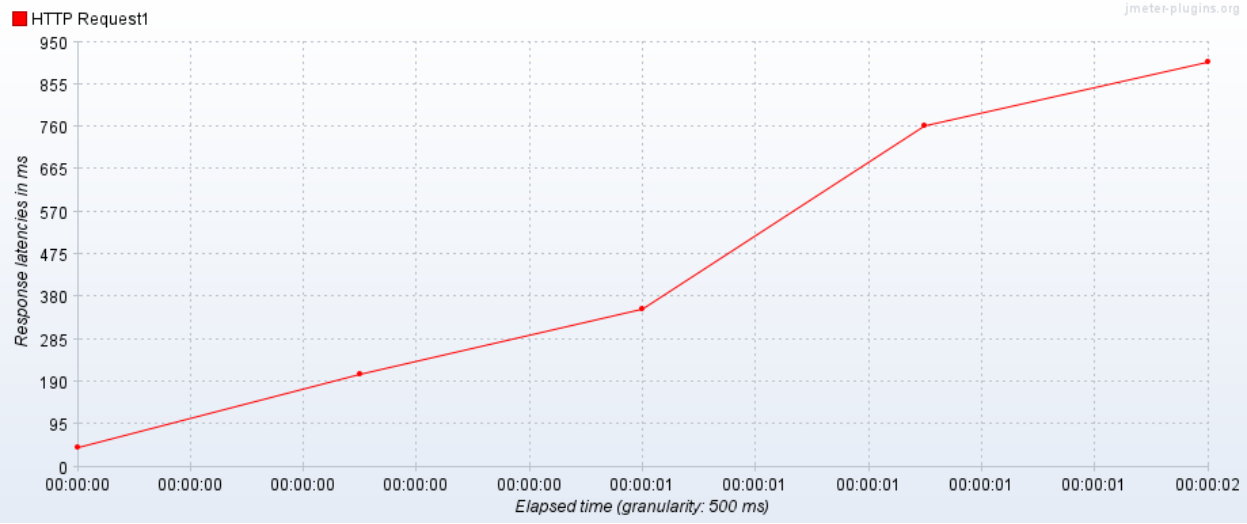
Bytes Throughput



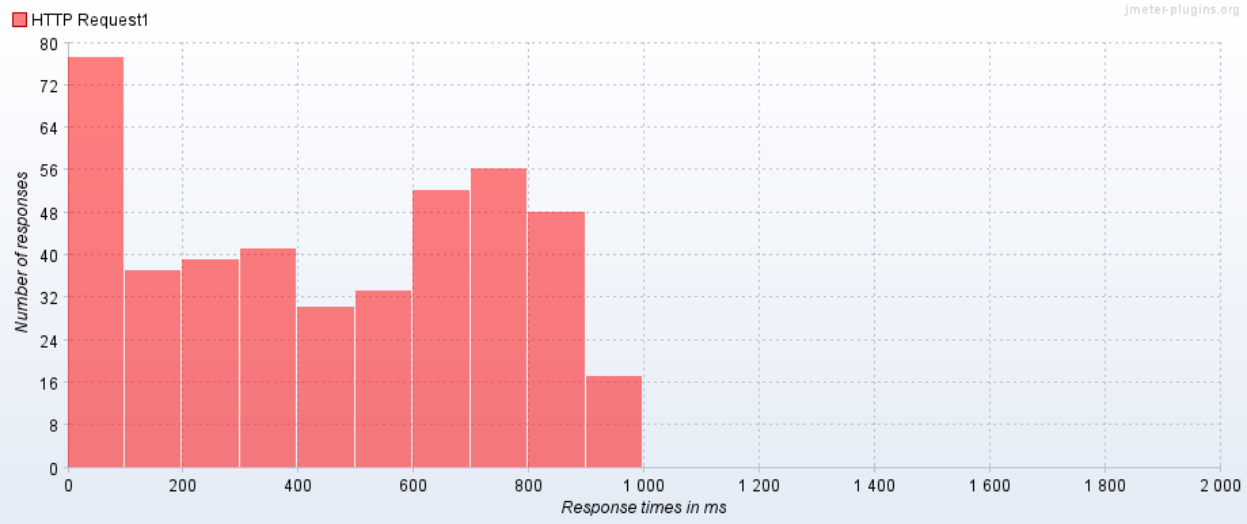
Response Codes per Second



Response Latencies Over Time



Response Times Distribution



3**) Login - Kubernetes(3 Replicas - D)**

Microservice – Dev Registry

Language - NodeJS

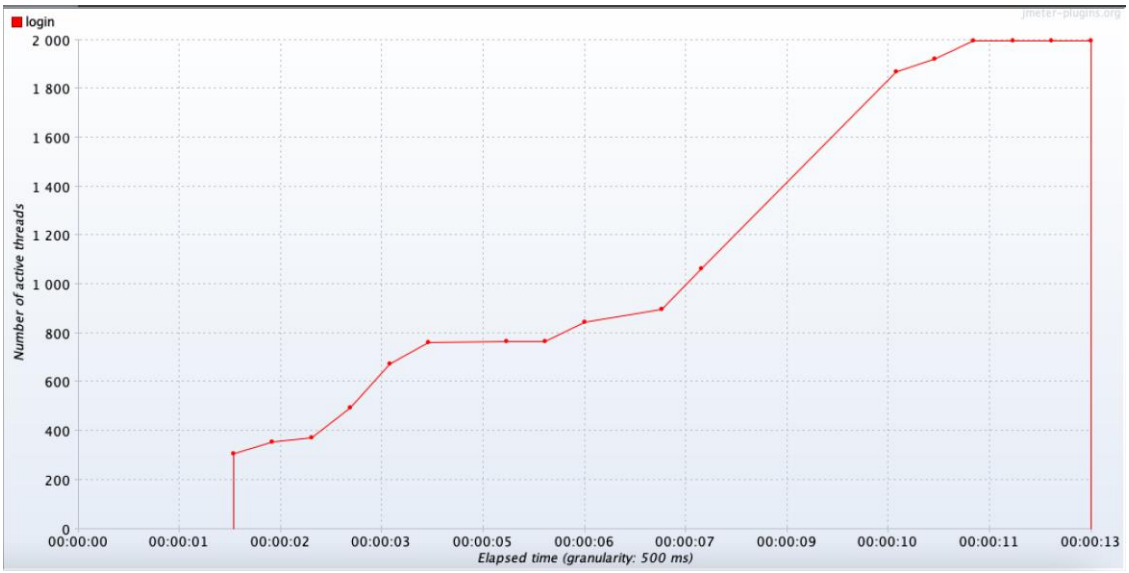
URL - /registry/api/v1/user/login

Max Number of users: 1998

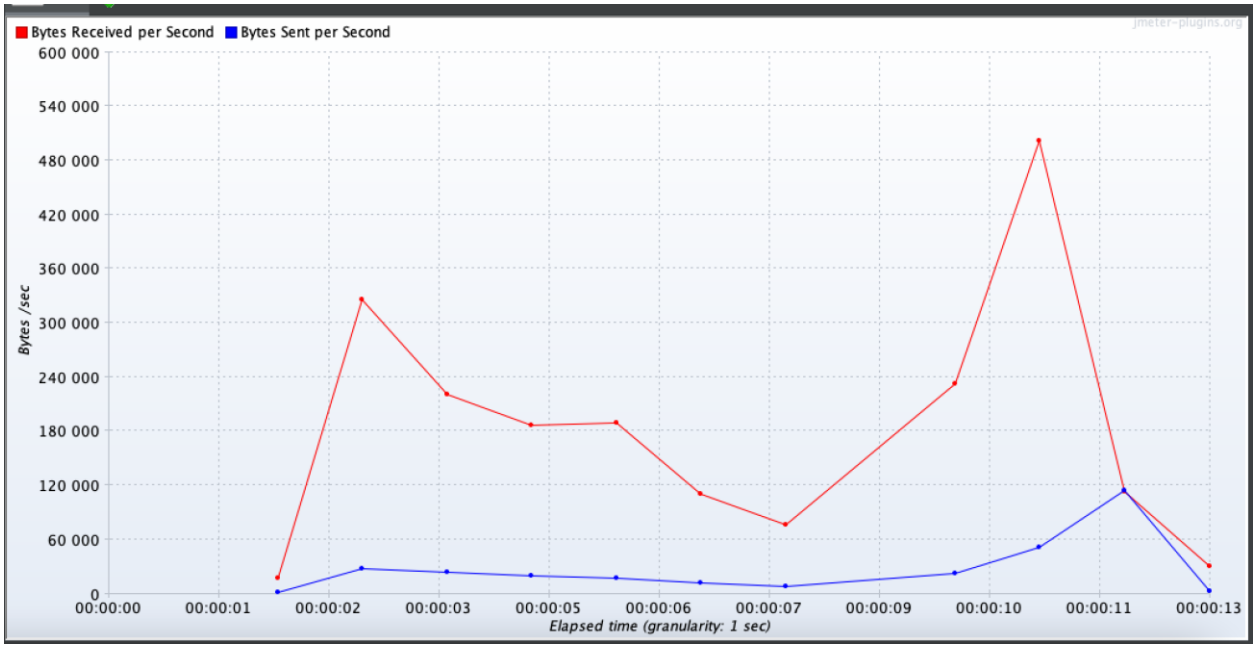
Error rate: 8.7%

Throughput: 400.7/sec

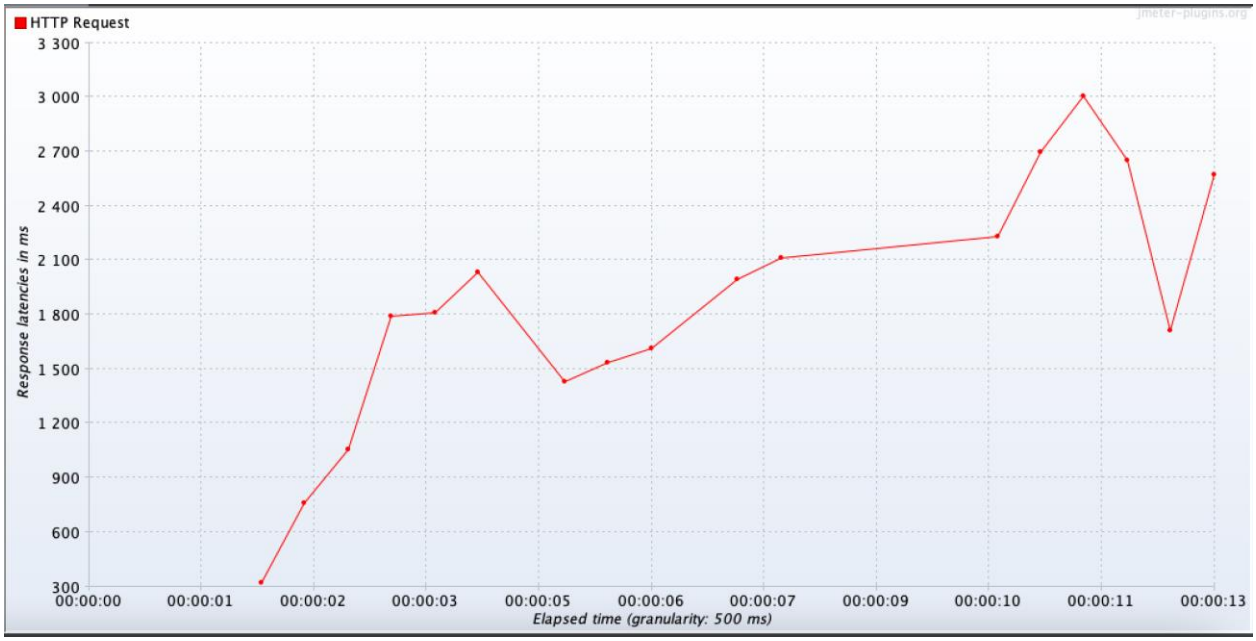
Active Threads



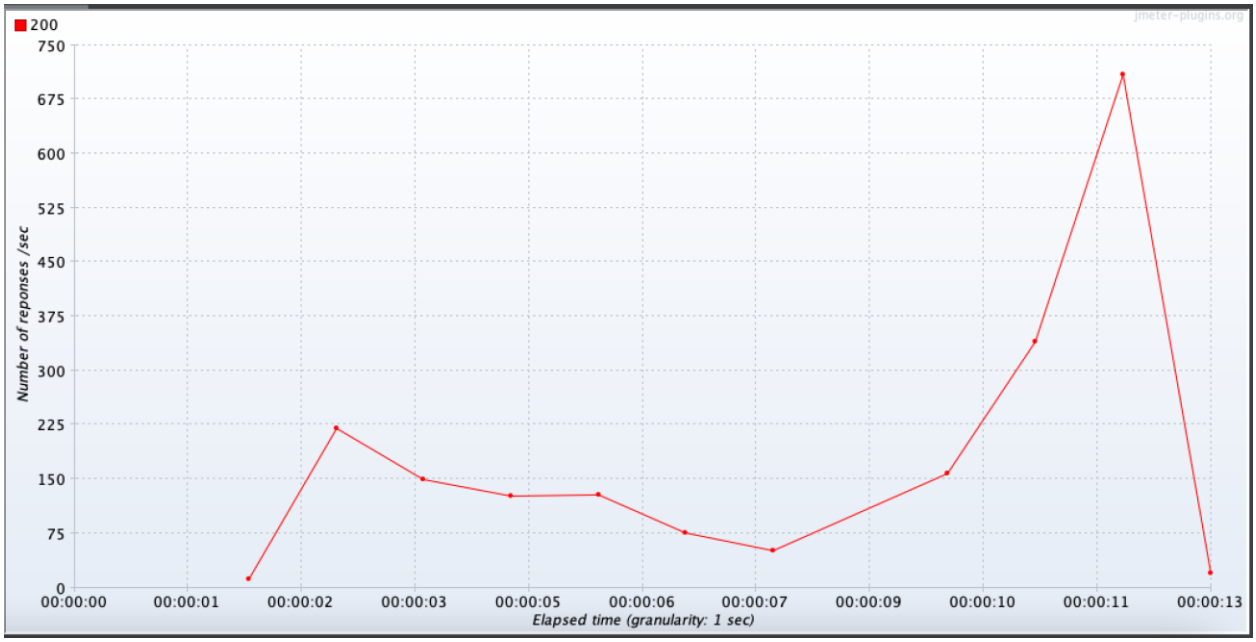
Bytes Throughput



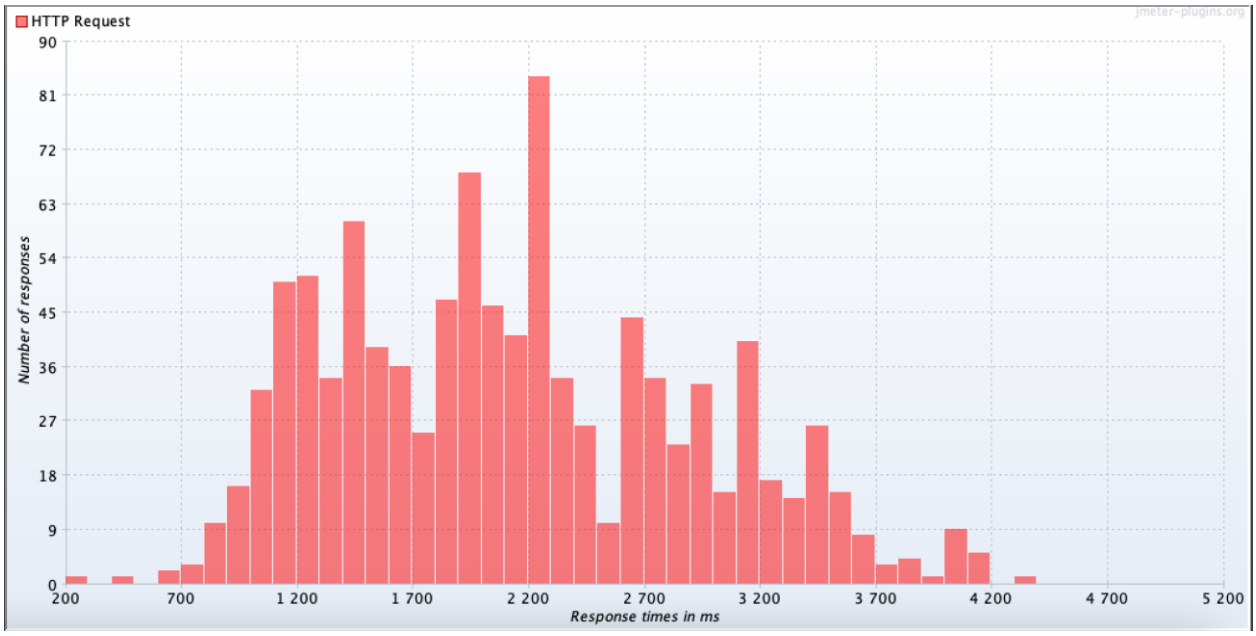
Response Latencies Over Time



Response Codes per Second



Response Times Distribution



4) Login - Kubernetes(5 Replicas - D)

Microservice – Dev Registry

Language - NodeJS

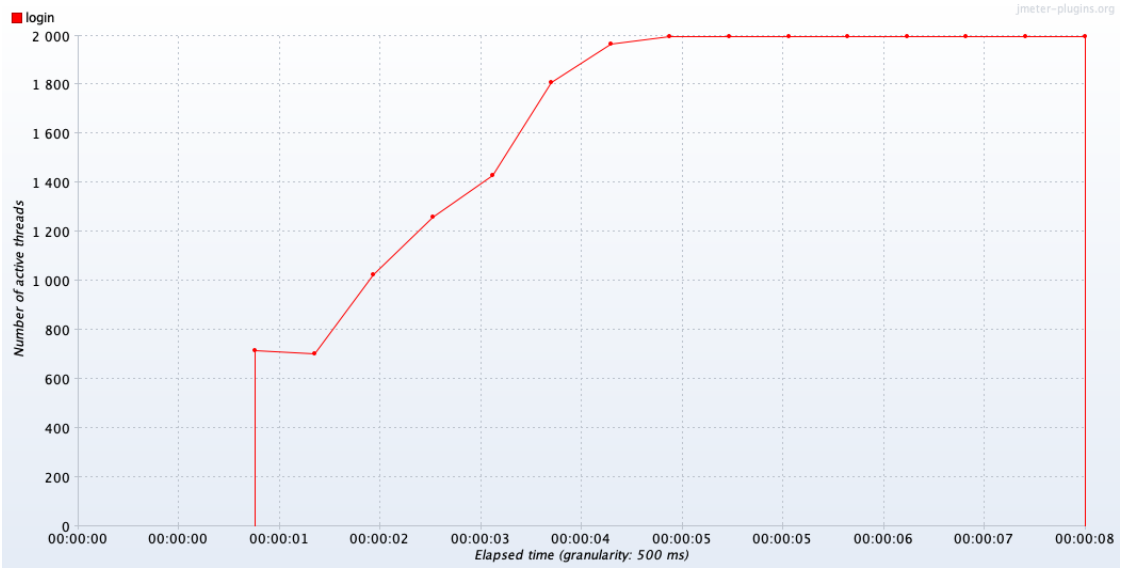
URL - /registry/api/v1/user/login

Max Number of users: 1997

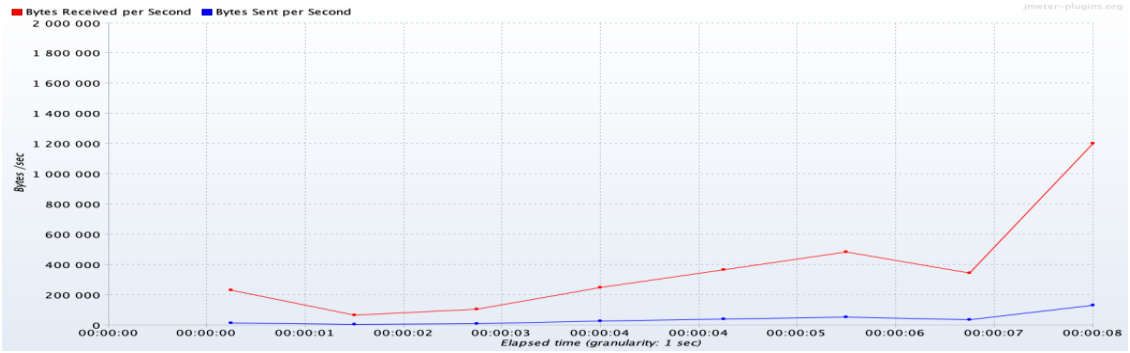
Error rate: 5.61%

Throughput: 269.8/sec

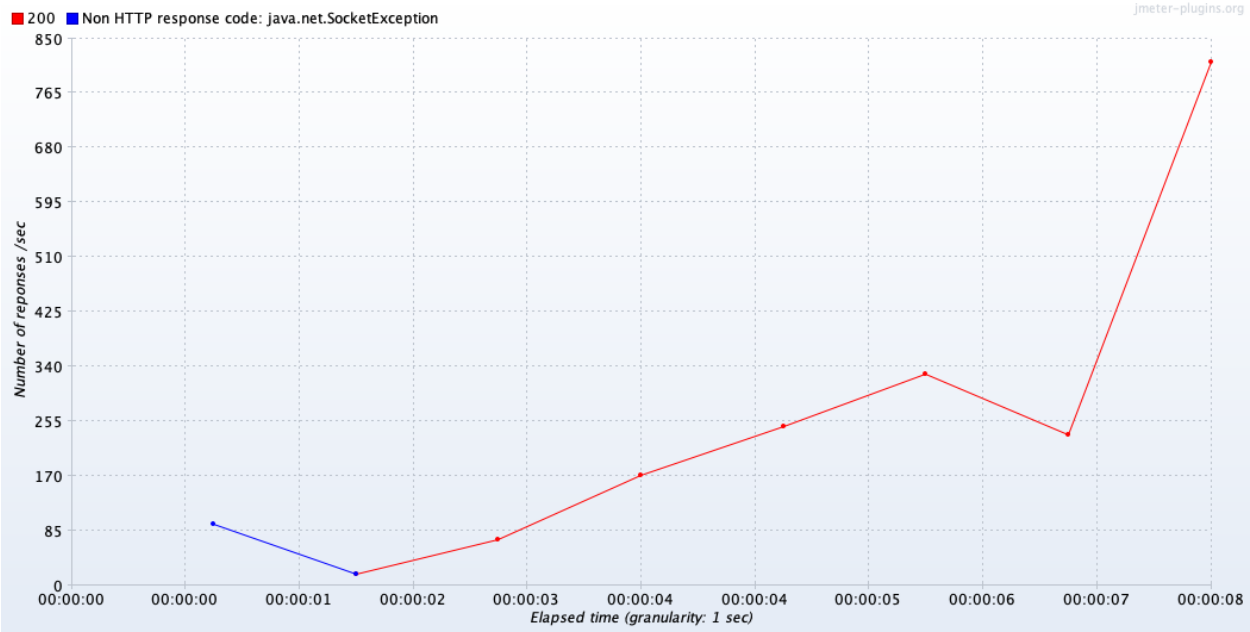
Active Threads



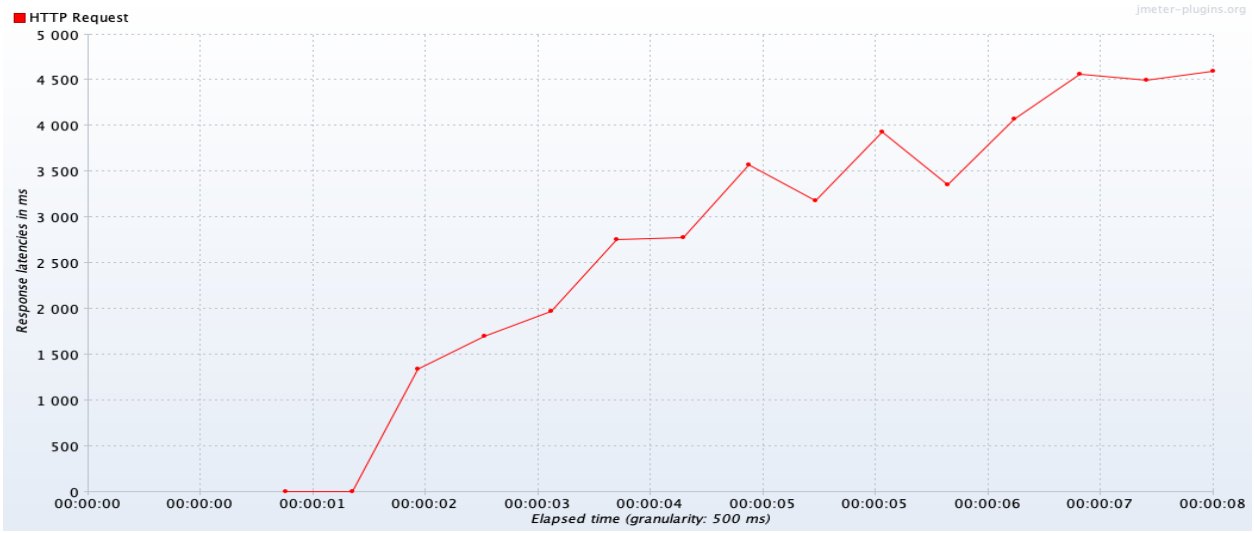
Bytes Throughput



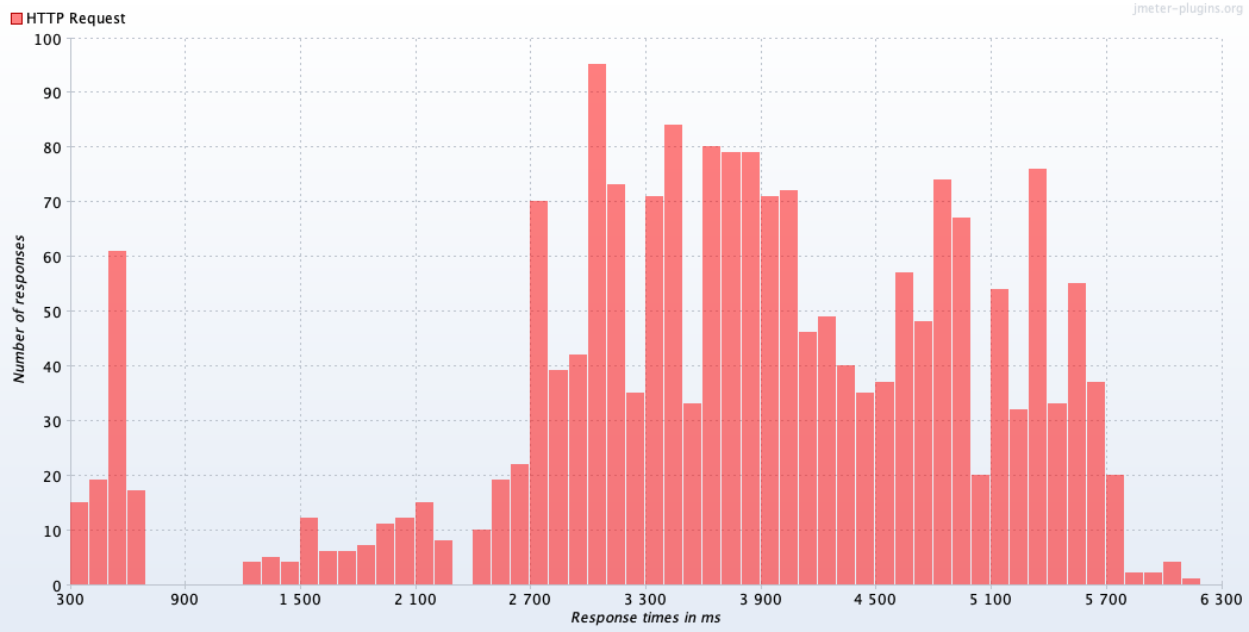
Response Codes per Second



Response Latencies Over Time



Response Times Distribution



Comparison between Login – Direct, Login – Gateway:

Name	Number of Users	Error Rate	Throughput
Login – Direct	580	9%	263.3/sec
Login - Gateway	430	11%	254.6/sec
Login-Kubernetes(3 Replicas-D)	1998	8.7%	400.7/sec
Login-Kubernetes(5 Replicas-D)	1997	5.61%	269.8/sec

Load handling capacity is more when the endpoint is reached directly as opposed to when we try to reach through the gateway. A 20% reduction in number of users is seen when we use the gateway with not a huge difference in the throughput. We also observe that the load handling capacity is more when we use Kubernetes as against when we hit the endpoint directly. It's even more when we use 5 replicas as compared to 3 replicas.

5) SignUp – Direct

Microservice – Dev Registry

Language - NodeJS

URL - /registry/api/v1/user/signUp

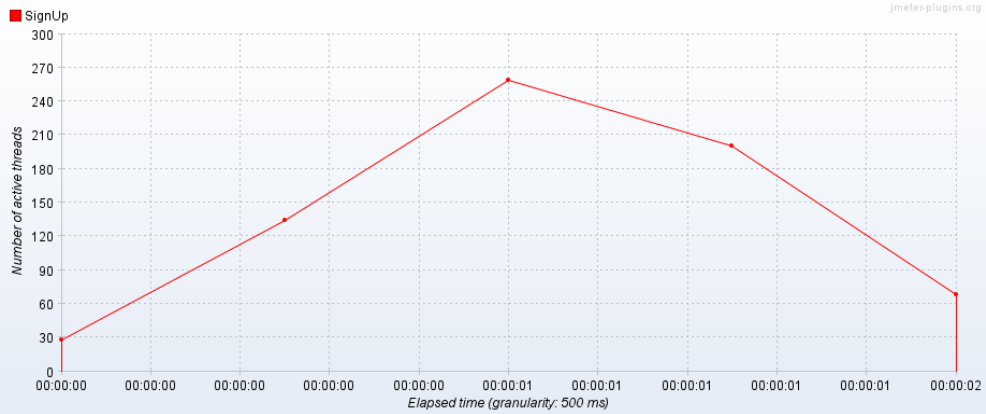
Max Number of users: 560

Error rate: 8.39%

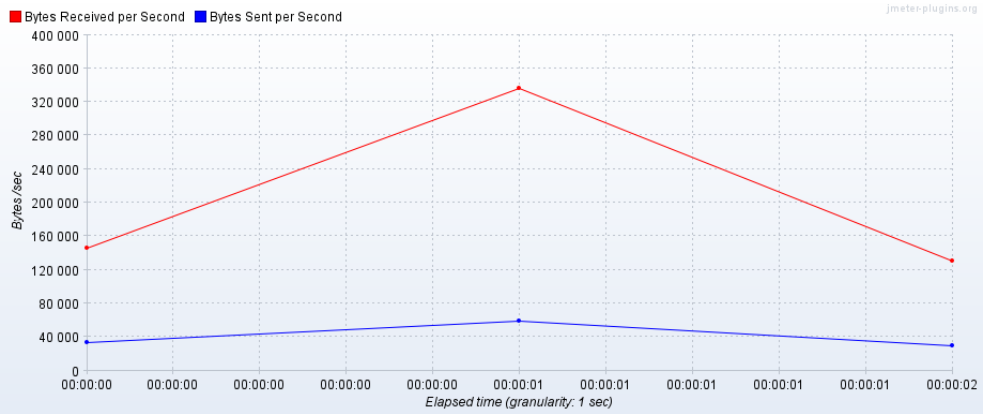
Throughput: 279.3/sec

Graph Results

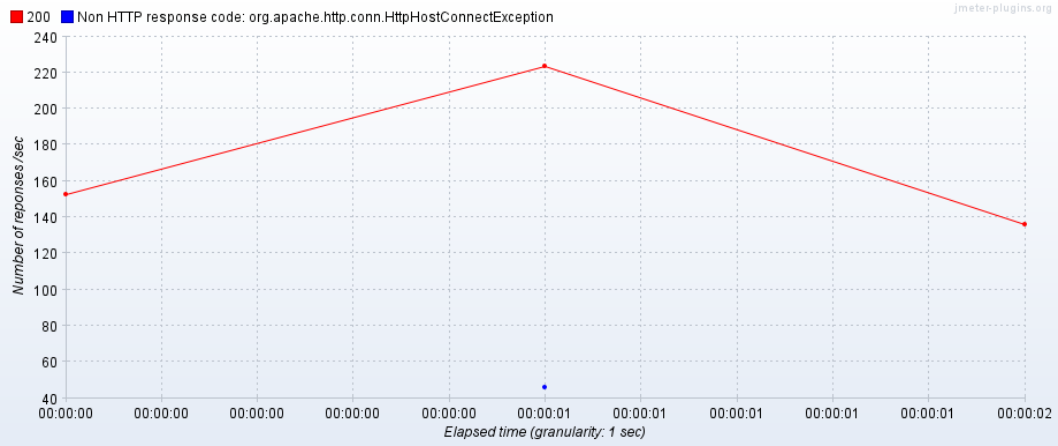
Active Threads



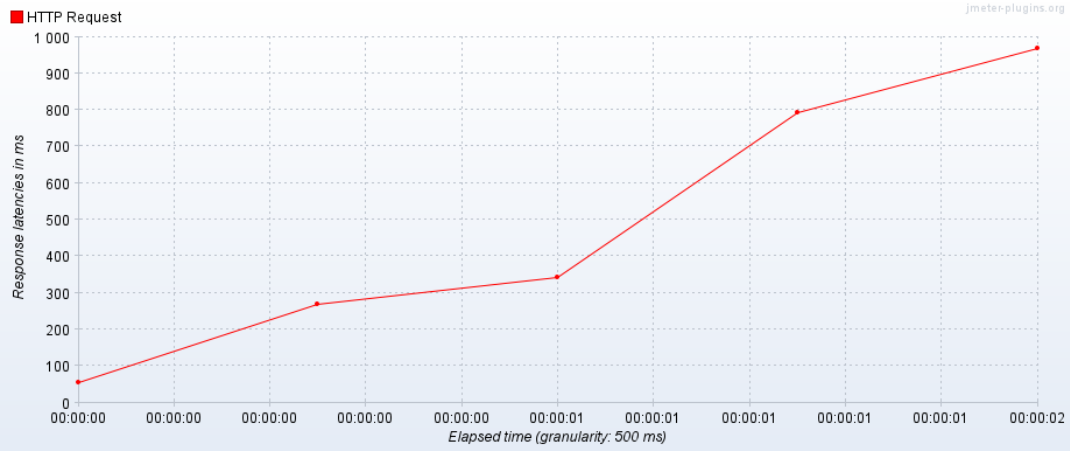
Bytes Throughput



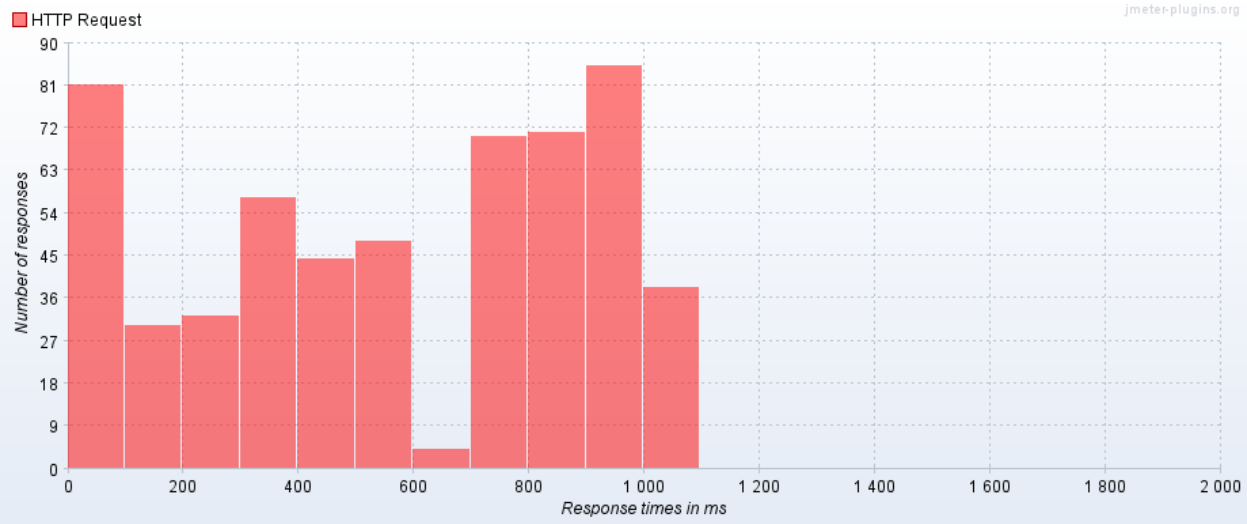
Response Codes Per Second



Response Latencies Over Time



Response Times Distribution



6**) SignUp – Gateway**

Microservice – Dev Registry

Language – Spring Boot, NodeJS

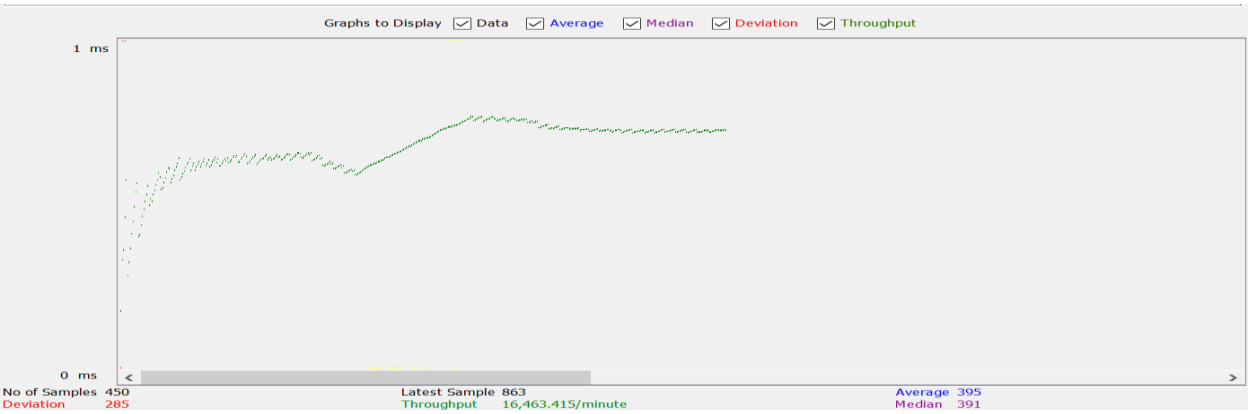
URL - /signUp

Max Number of users: 450

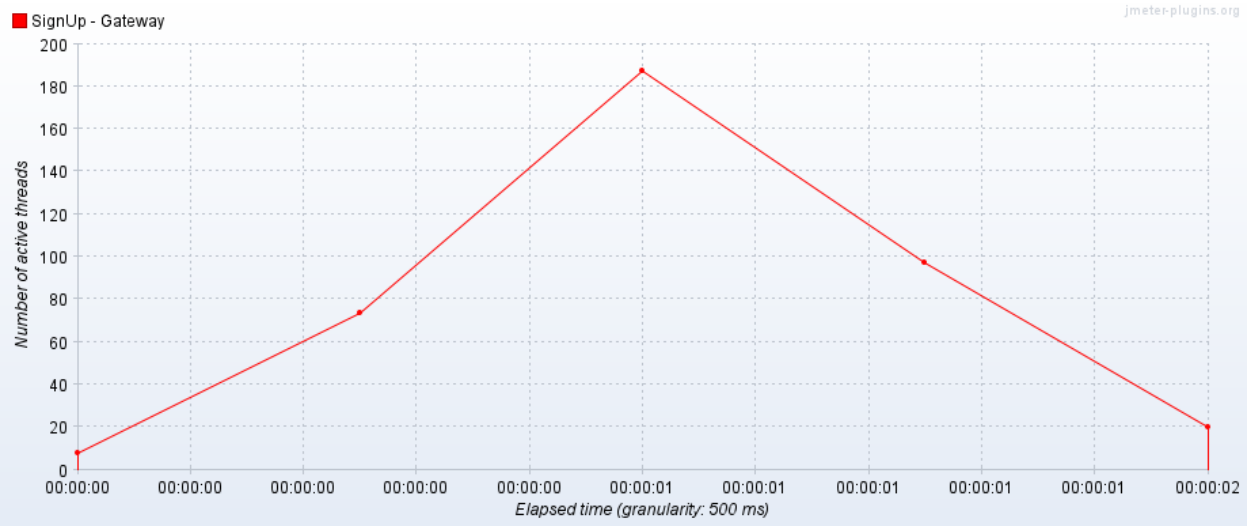
Error rate: 13.11%

Throughput: 274.4/sec

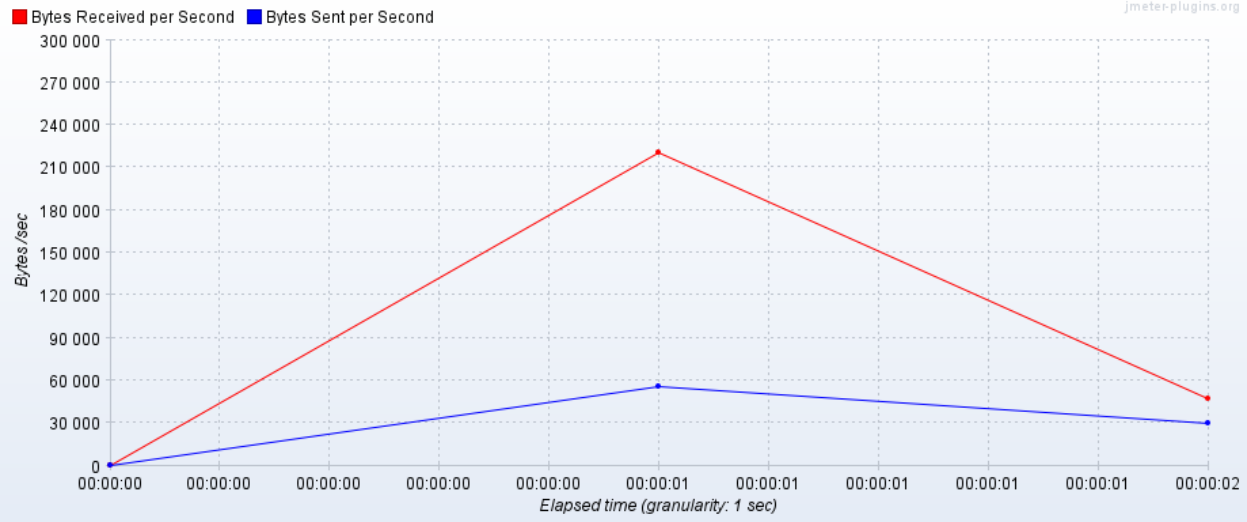
Graph Results



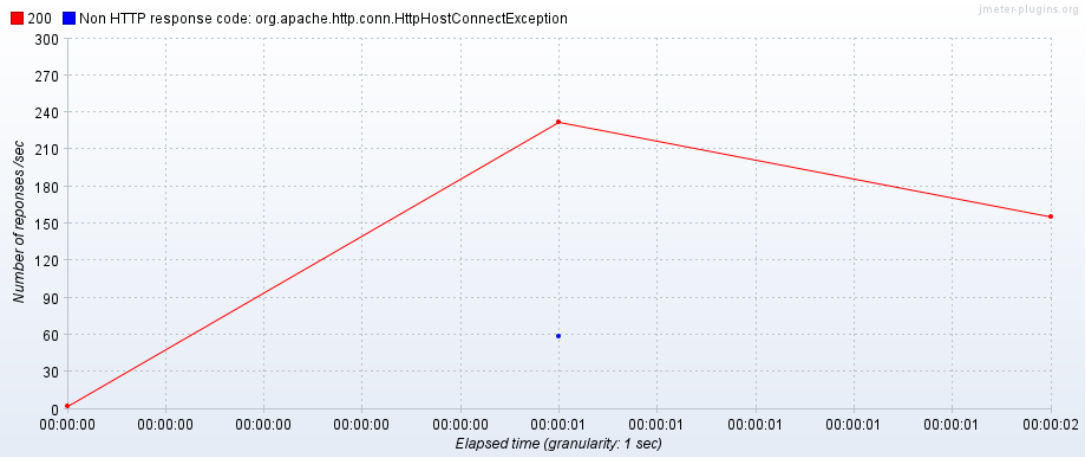
Active Threads



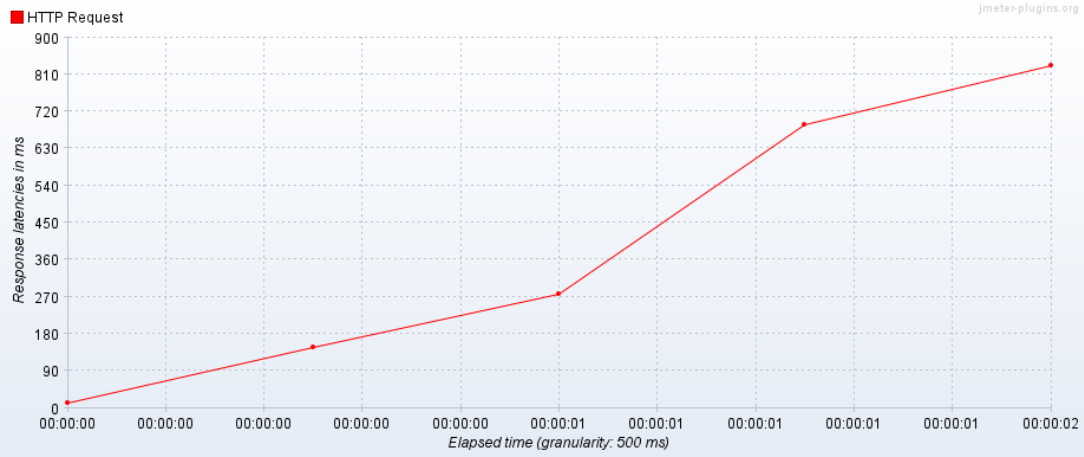
Bytes Throughput



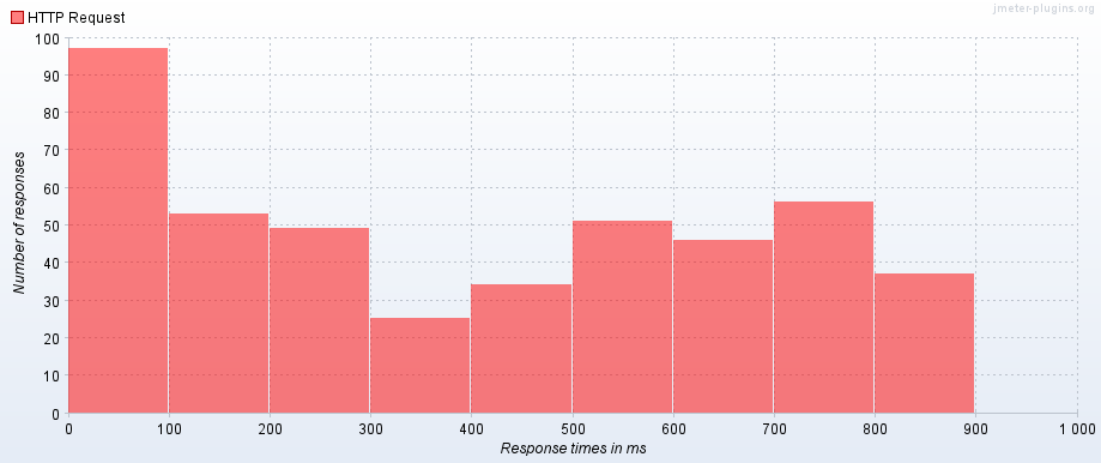
Response Codes per Second



Response Latencies over Time



Response Times Distribution



7**) SignUp – Kubernetes With Load Balancing 3 Replicas(D)**

Microservice – Dev Registry

Language - NodeJS

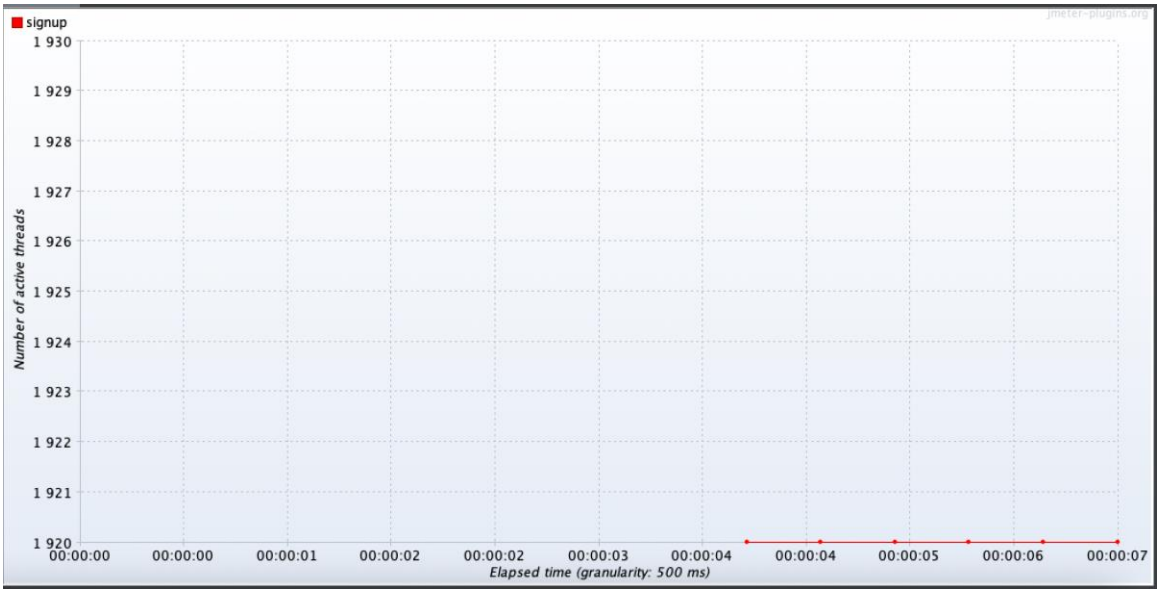
URL - /registry/api/v1/user/signUp

Max Number of users: 1920

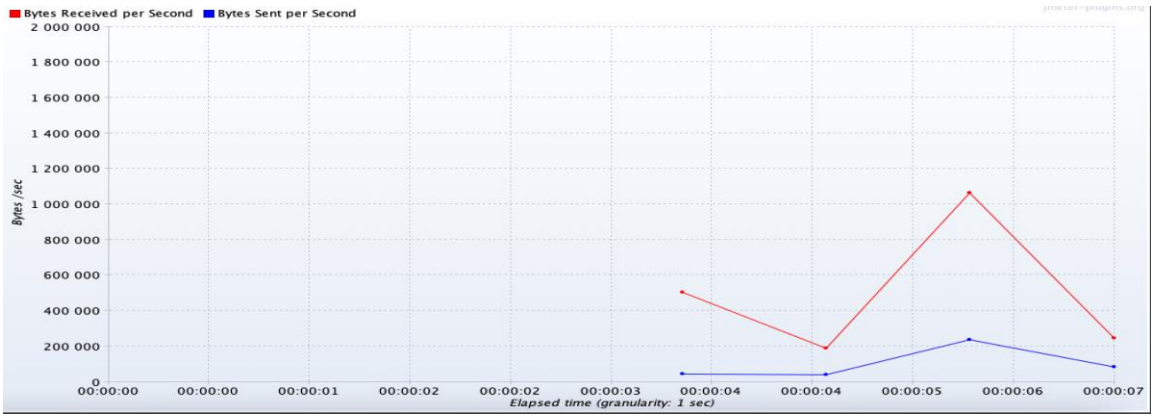
Error rate: 10.78%

Throughput: 438.5.3/sec

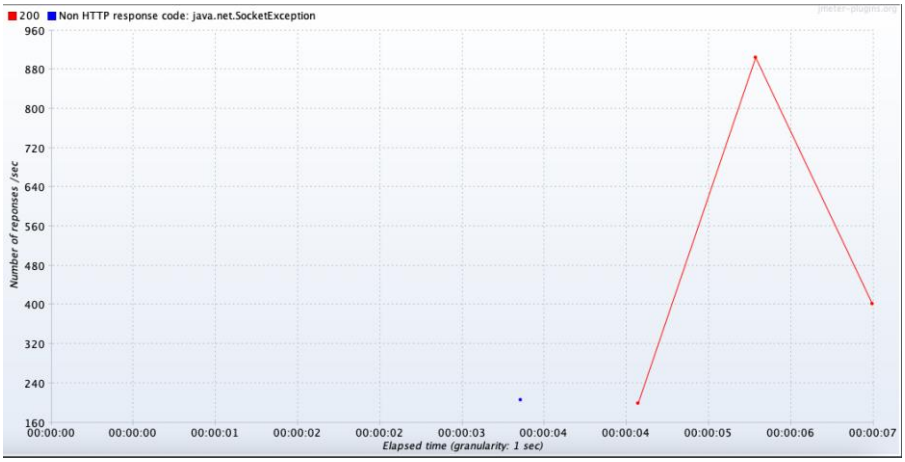
Active Threads



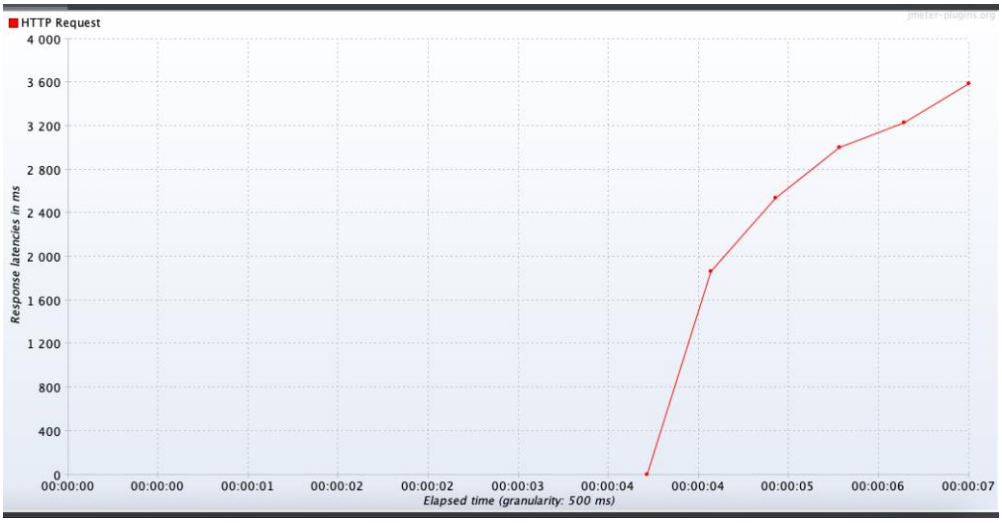
Bytes Throughput



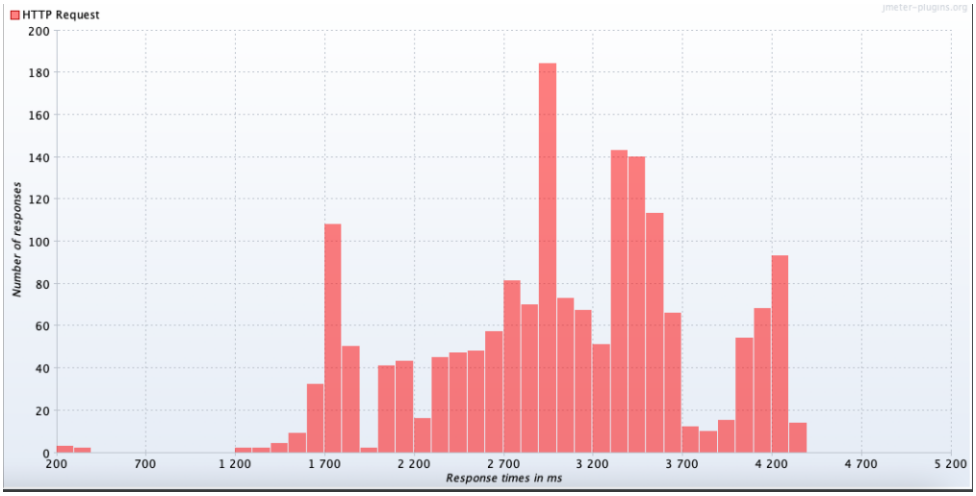
Response Codes per Second



Response Latencies over Time



Response Times Distribution



8**) SignUp – Kubernetes With Load Balancing 5 Replicas(D)**

Microservice – Dev Registry

Language - NodeJS

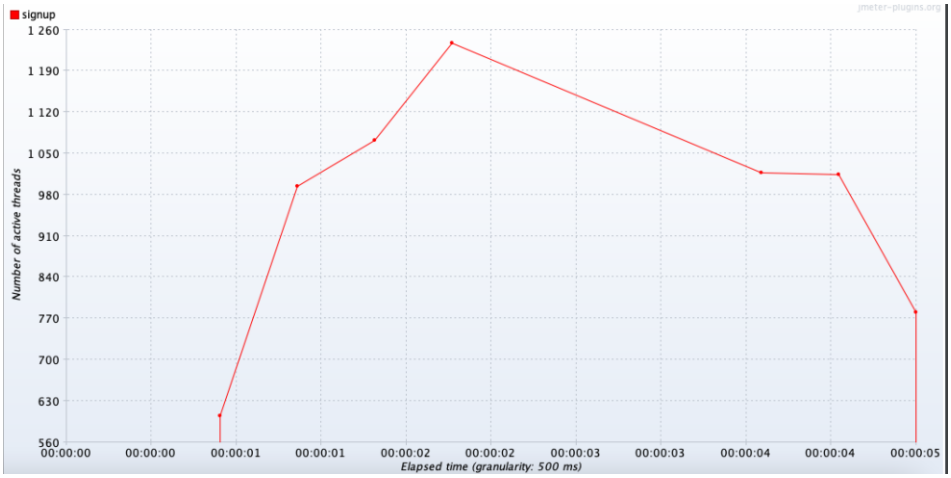
URL - /registry/api/v1/user/signUp

Max Number of users: 2220

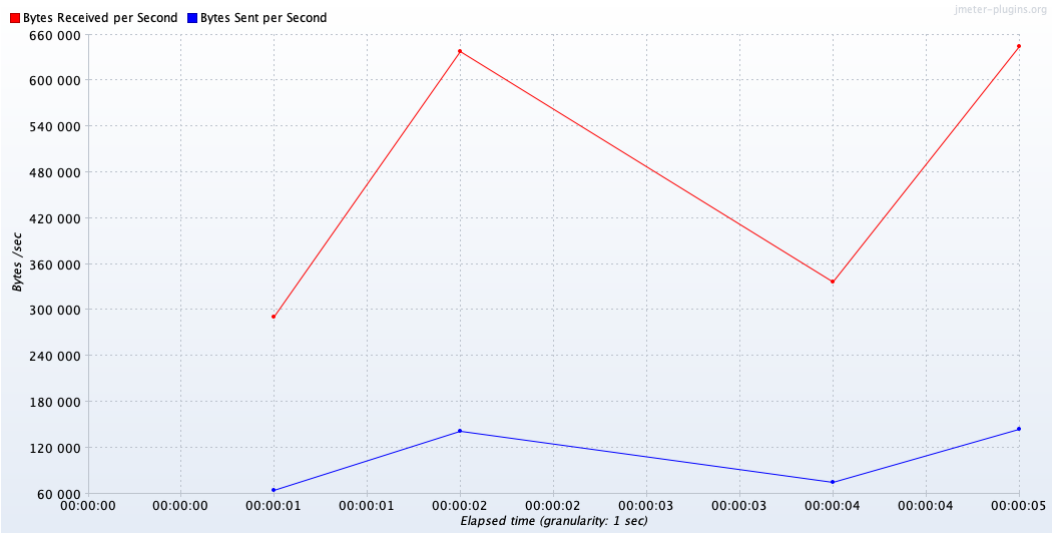
Error rate: 12.48%

Throughput: 395.0/sec

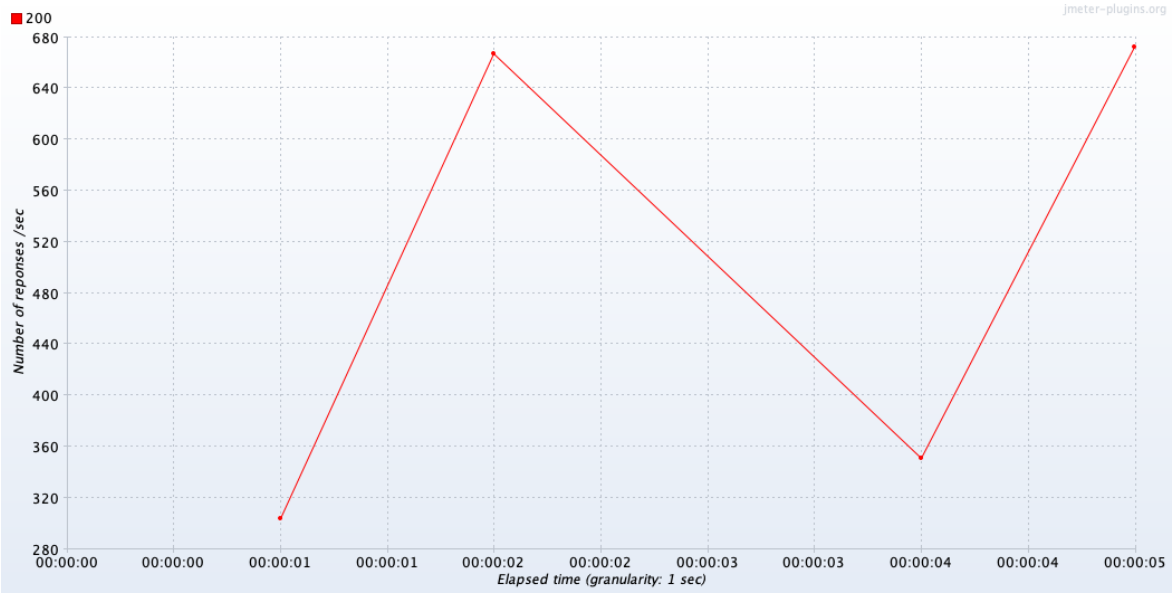
Active Threads



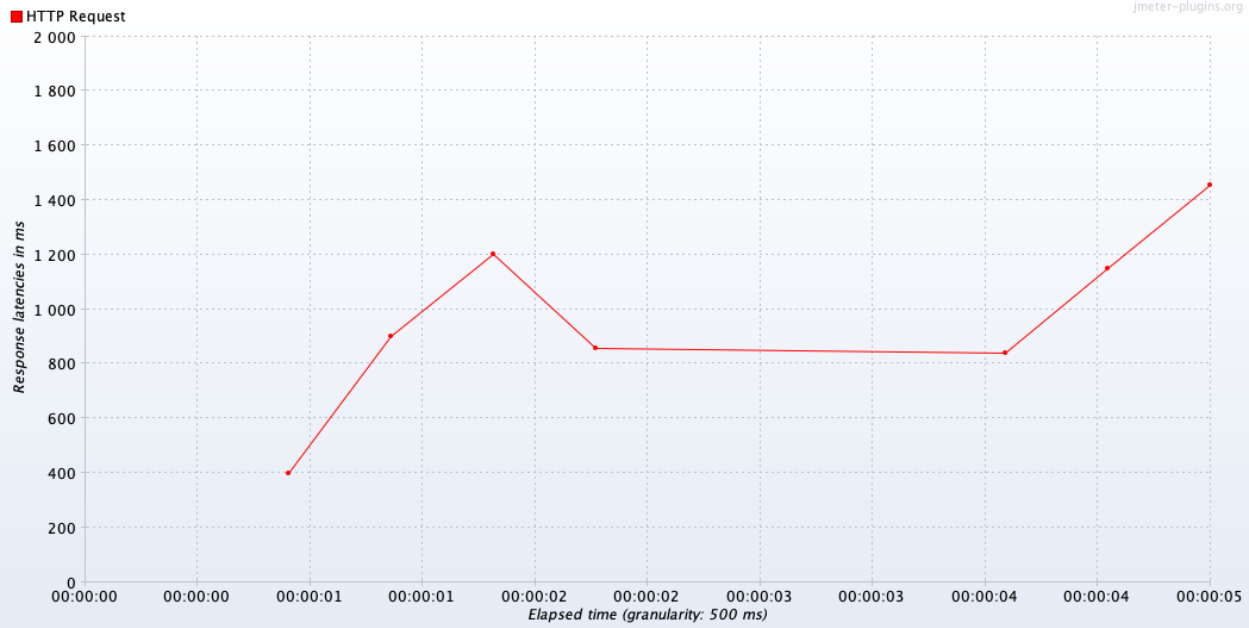
Bytes Throughput



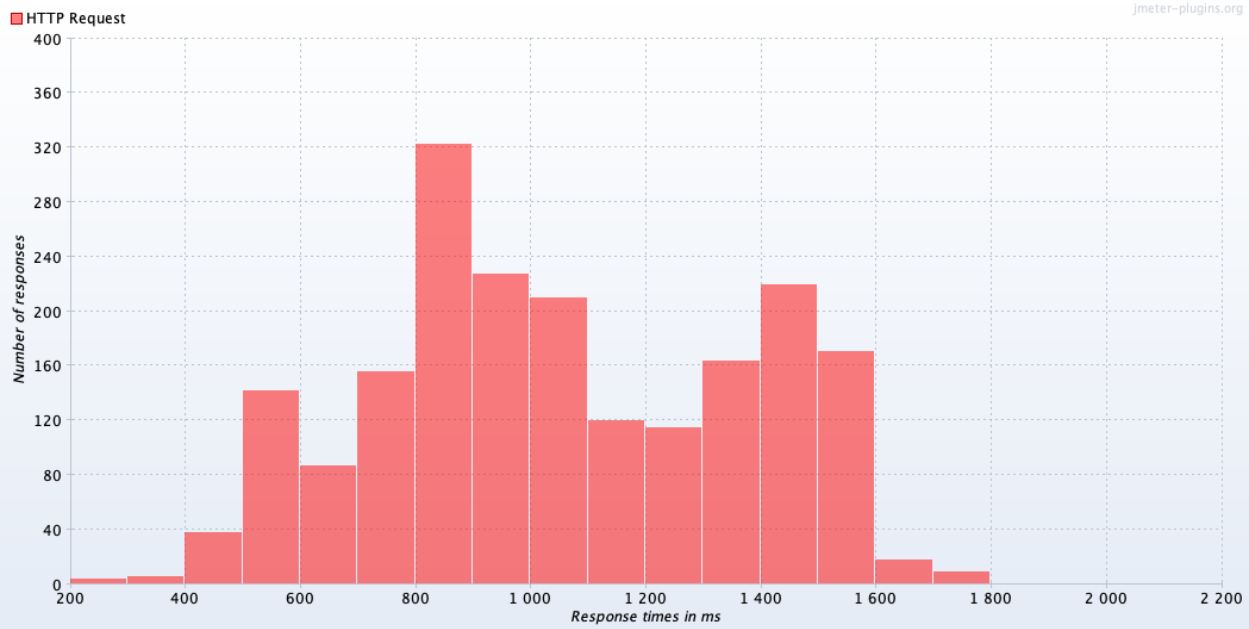
Response Codes per Second



Response Latencies over Time



Response Times Distribution



Comparison between SignUp – Direct, SignUp – Gateway, Kubernetes 3 Replicas:

Name	Users	Error Rate	Throughput
SignUp – Direct	560	8.39%	279.3/sec
SignUp - Gateway	450	13.11%	274.4/sec
SignUp - Kubernetes(3 Replicas - D)	1920	10.78%	126.83/sec
SignUp - Kubernetes(5 Replicas - D)	2220	12.48%	395.0/sec

More users can reach the Signup endpoint directly as compared to reaching through the Gateway. The difference in throughput is minimal. We also observe that the load handling capacity is more when we use Kubernetes as against when we hit the endpoint directly. It's even more when we use 5 replicas as compared to 3 replicas.

9**) Plotting – Direct**

Microservice – Plotting

URL - /getPlottedData

Language – Python Flask

Max Number of users: 46

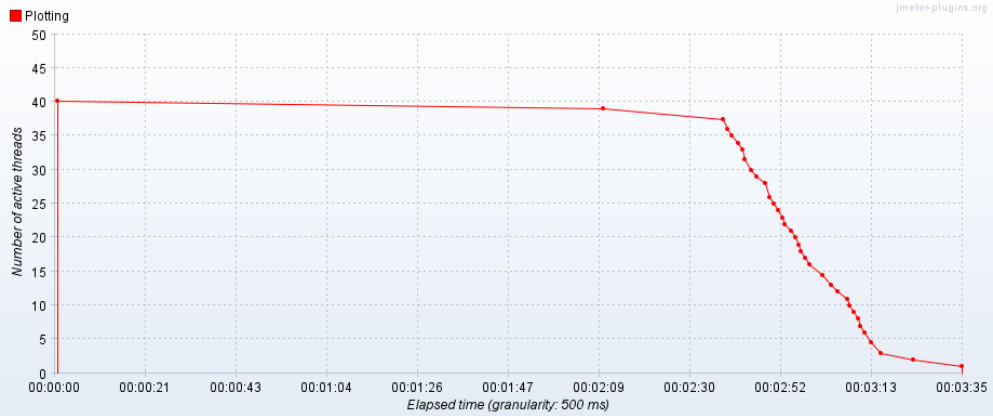
Error rate: 0%

Throughput: 12.5/min

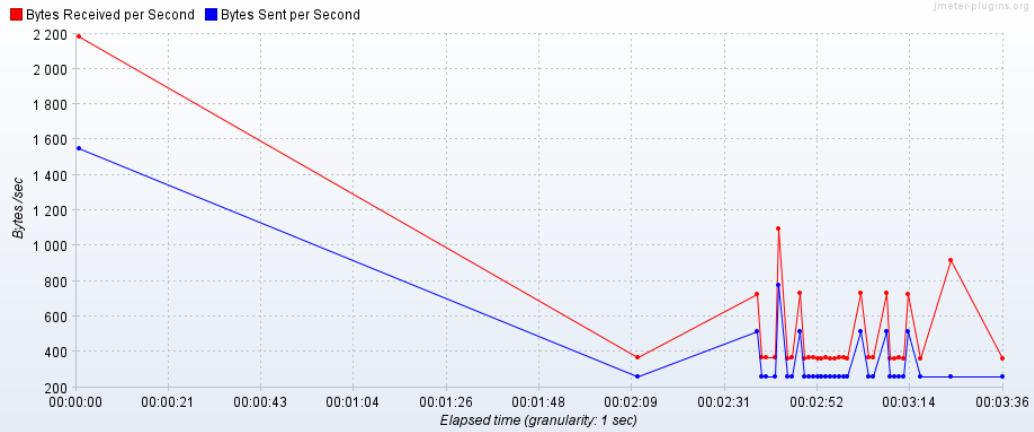
Acceptable Users: 45

We observed that our system stopped responding when we crossed **the 45 user mark** due to excessive CPU utilization. The error rate and throughput values are for 45 users.

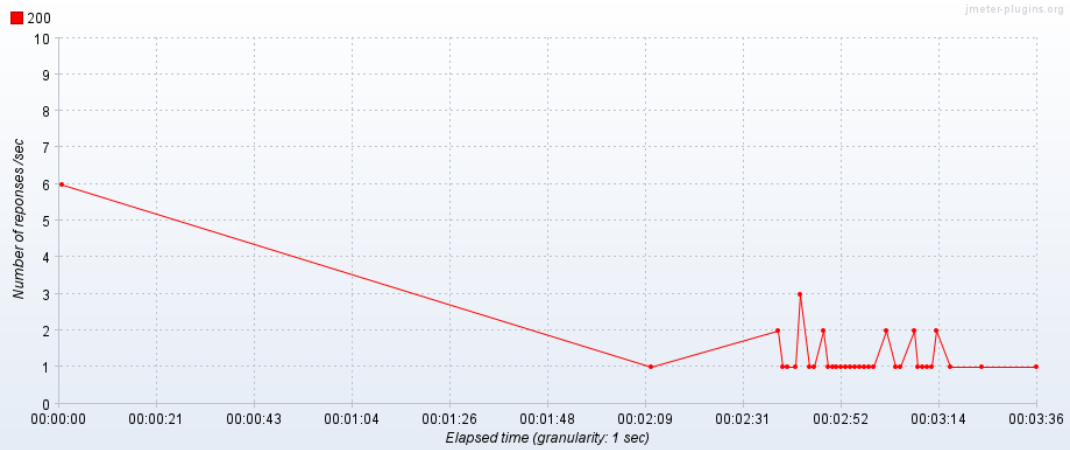
Active Threads



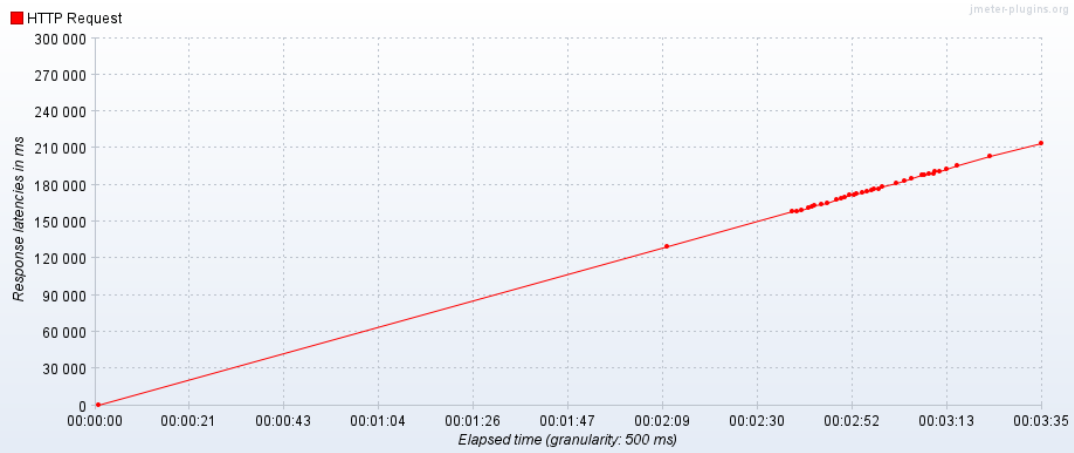
Bytes Throughput



Response Codes Over Time



Response Latencies Over Time



10**) Plotting – Gateway**

Microservice – Plotting

URL - /plotting

Language – Python Flask, Spring Boot

Max Number of users: 36

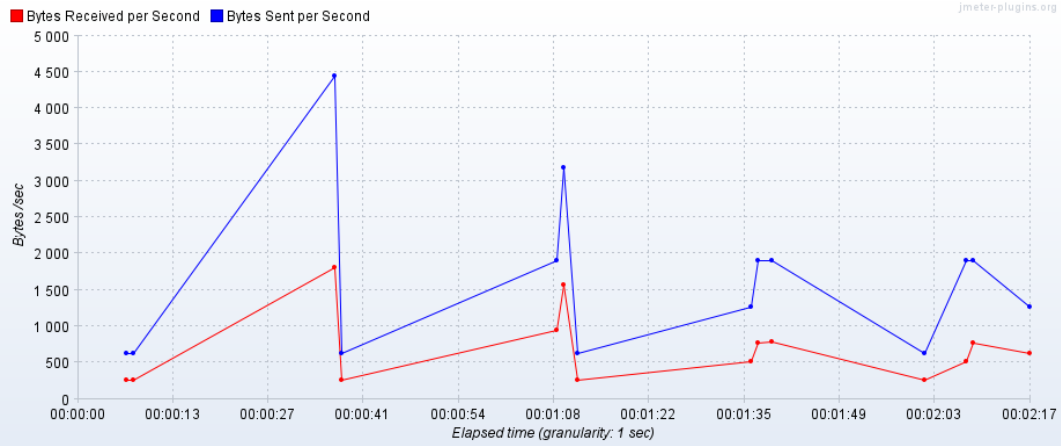
Error rate: 28%

Throughput: 15.8/min

The system stopped responding when the **threshold of 36** crossed because of excessive memory utilization. The error rate and throughput values are for 36 users.

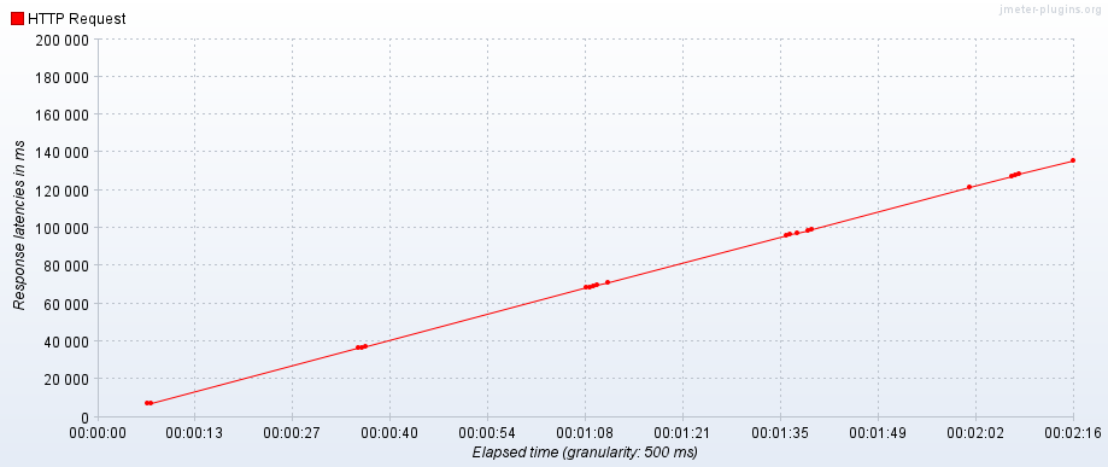
Active Threads

Bytes Throughput



Response Codes Over Time

Response Latencies



11) Microservice – Plotting - Kubernetes (3 replicas)

Microservice – Plotting

URL - /getPlottedData

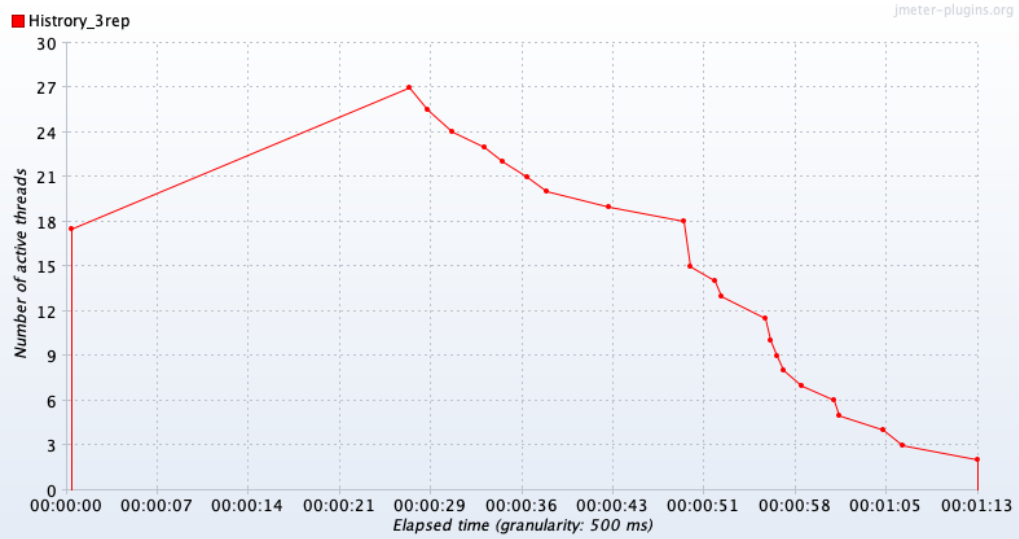
Language – Python Flask

Max Number of users: 40

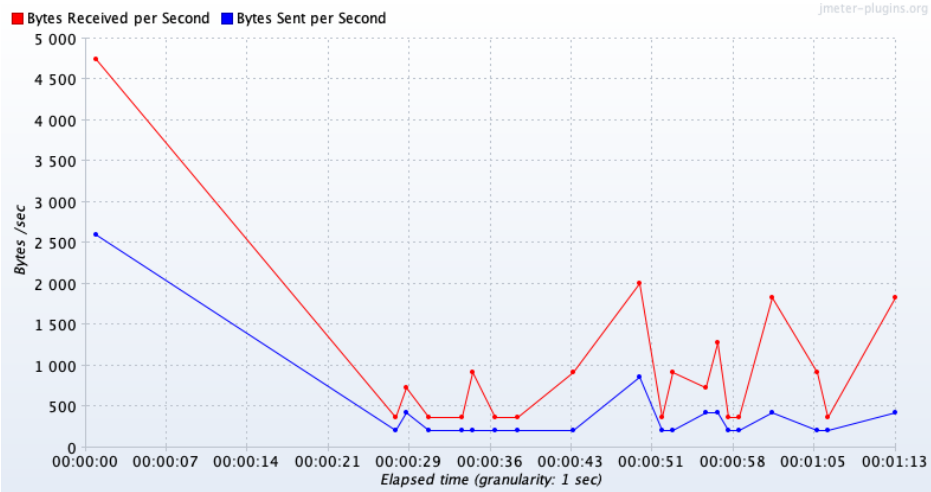
Error rate: 0%

Throughput: 32.8/min

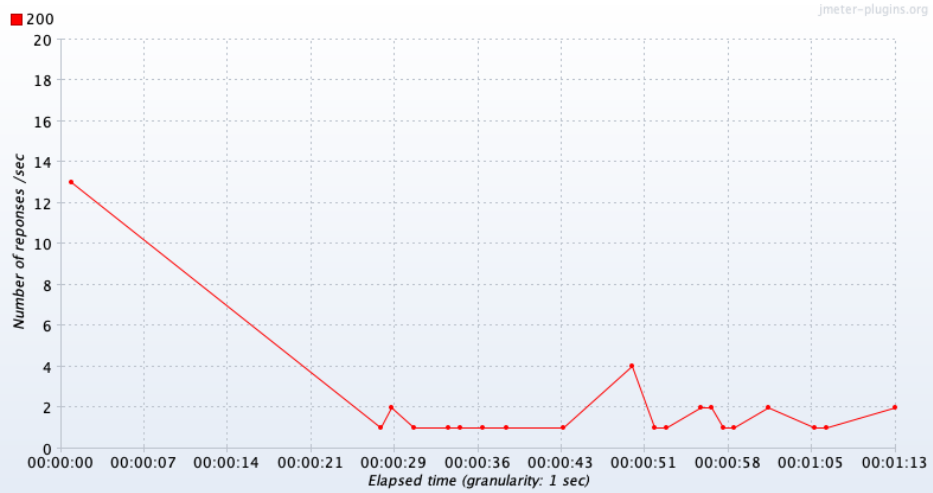
Active Threads



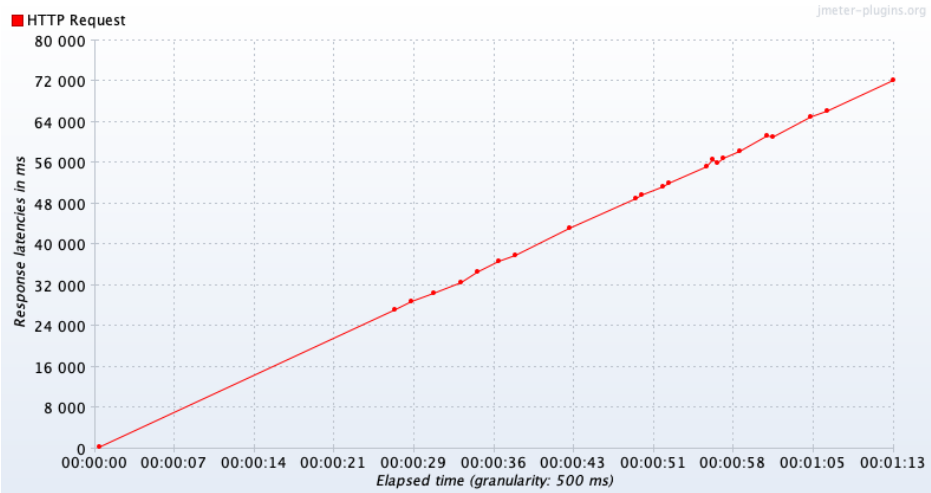
Bytes Throughput



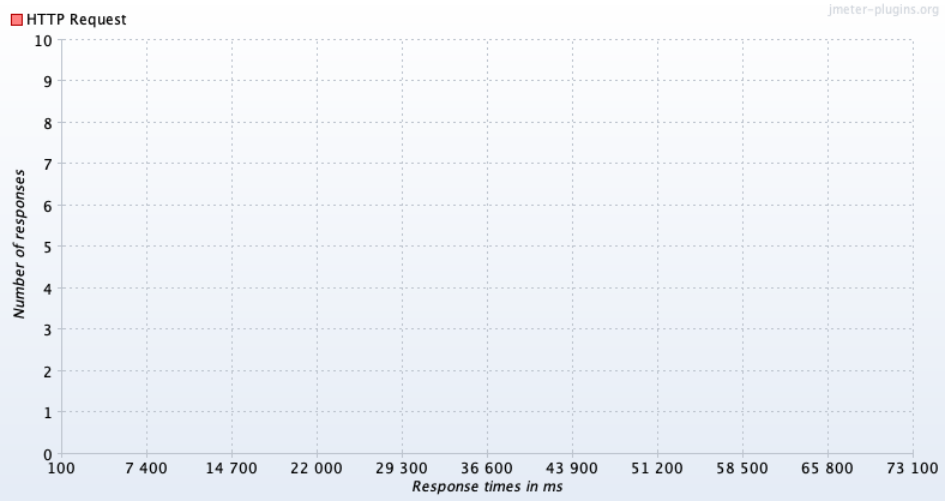
Response Codes Over Time



Response Latencies



Response Times Distribution



12) Microservice – Plotting - Kubernetes - (5 replicas)

Microservice – Plotting

URL - /getPlottedData

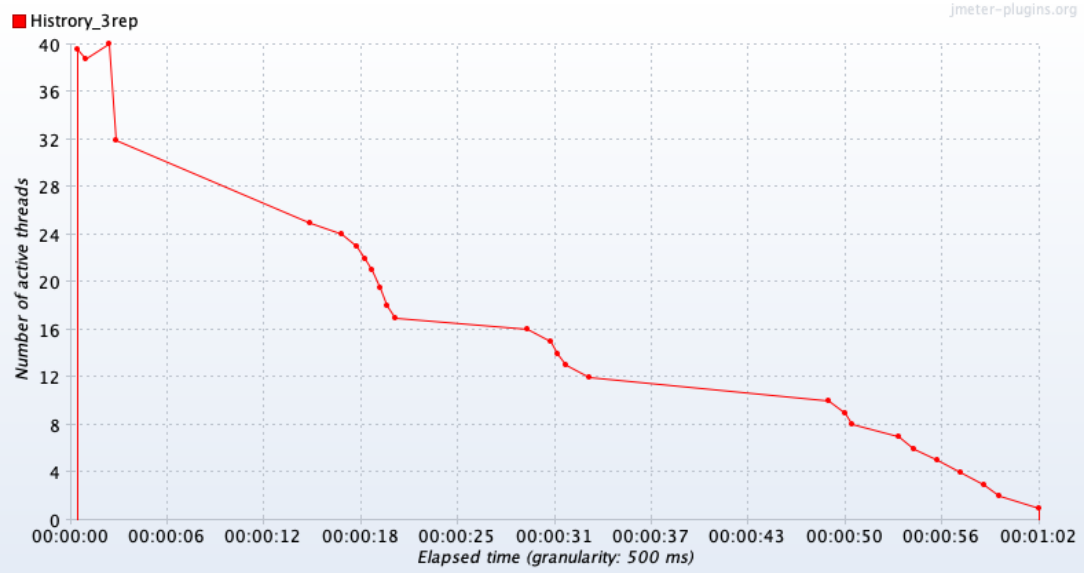
Language – Python Flask

Max Number of users: 50

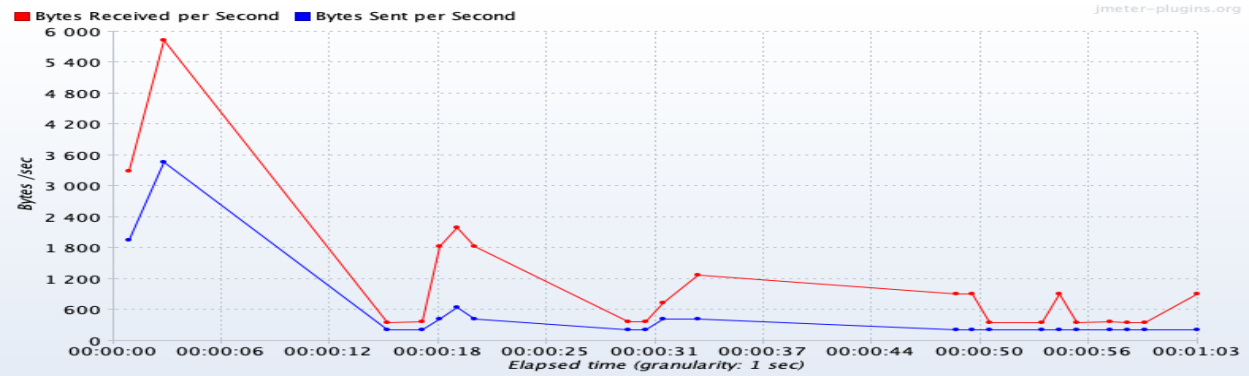
Error rate: 0%

Throughput: 47.9/min

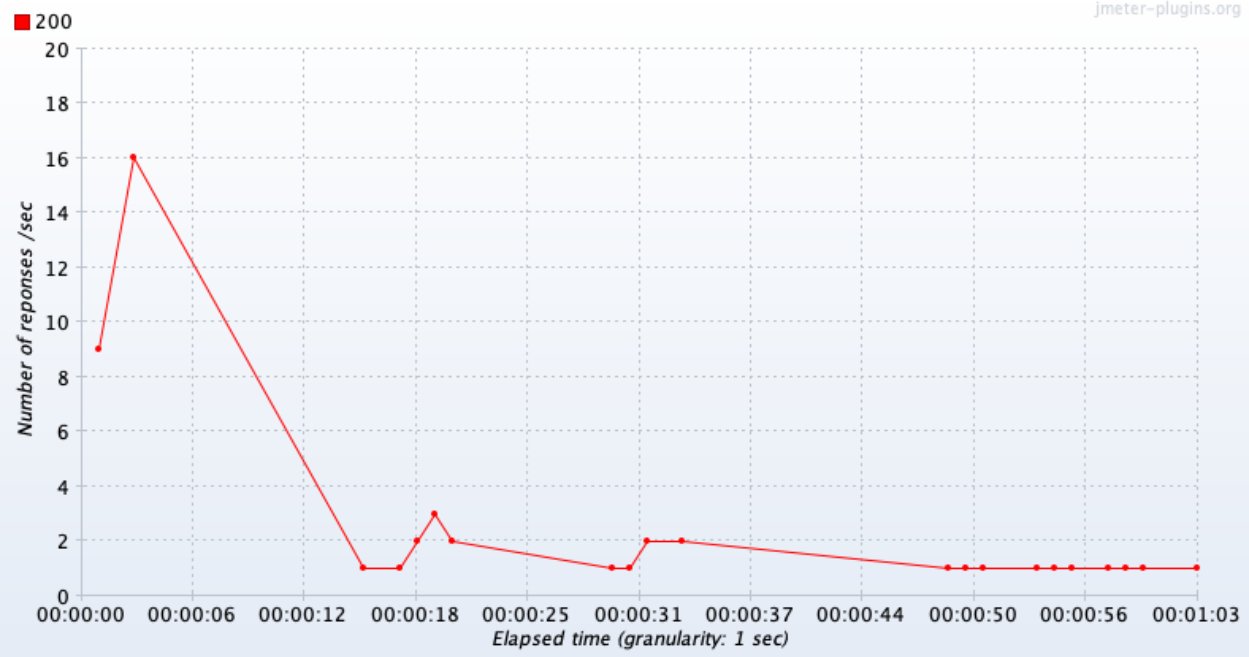
Active Threads



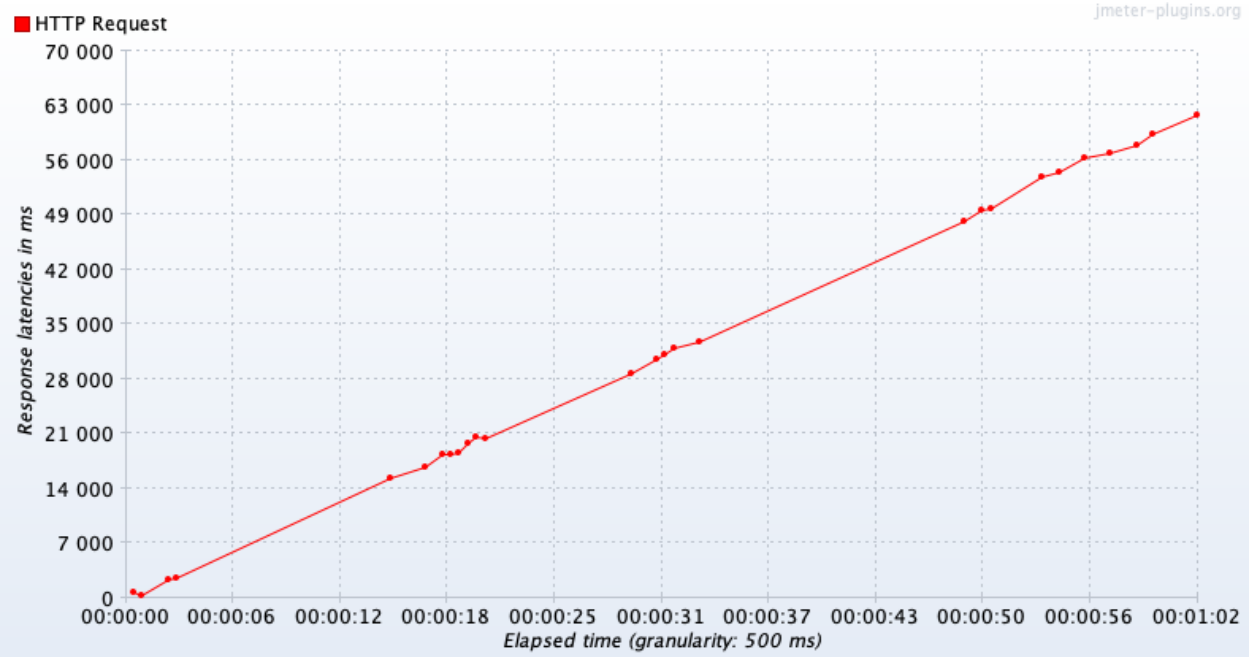
Bytes Throughput



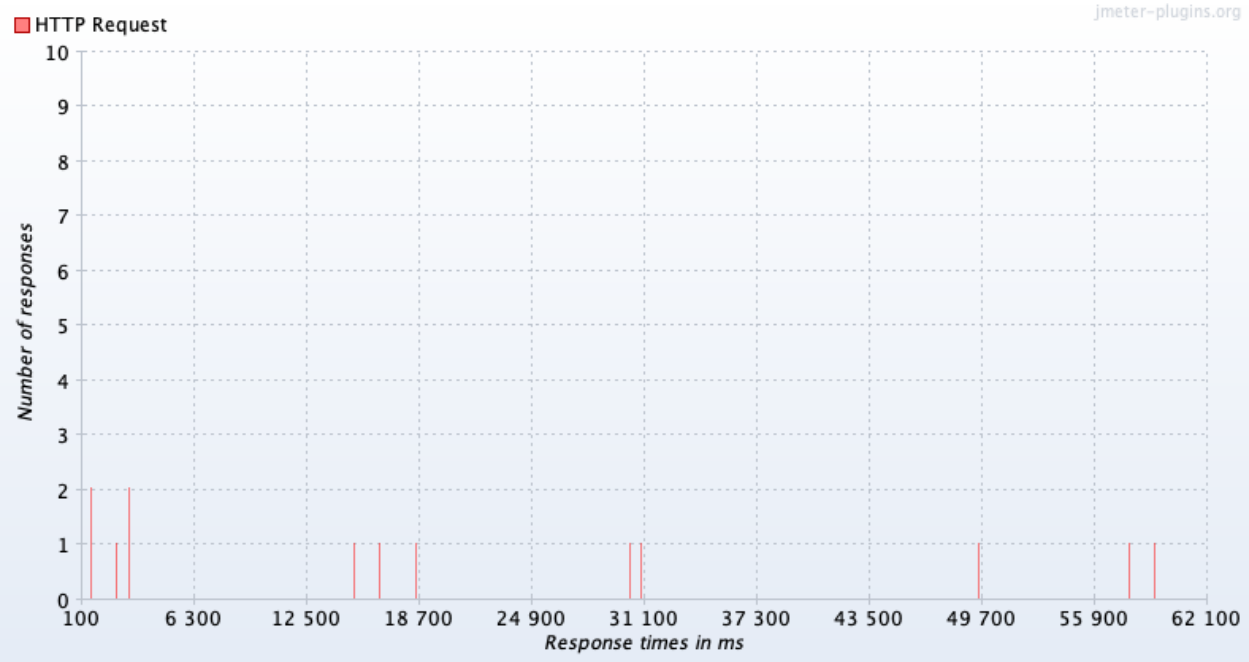
Response Codes Over Time



Response Latencies



Response Times Distribution



Comparison between Plotting – Direct, Plotting – Gateway:

Name	**User Break-point**	**Max Users**	**Error Rate**	**Throughput (Max)**
Plotting – Direct	46	45	0%	12.5/min
Plotting - Gateway	37	36	28%	15.8/min
Plotting - Kuber-netes(3 Replicas - D)	40	40	0%	32.8/min
Plotting- Kubernetes(5 Replicas - D)	50	50	0%	47.9/min

Plotting is a resource intensive operation because we are first plotting the image based on the user inputs, temporarily storing the image locally and then uploading it to the Cloudinary server. Hence, we notice that the load handling here is quite less as compared to the other microservices. When we reach the

endpoint directly, more users are able to access it. However, the throughput is higher when we reach the endpoint through the gateway and it's likely because of the lower user count that can be accommodated. We didn't see much of a difference when we tried using Kubernetes with 3/5 replicas because of memory leakage while executing the plotting operation. However, when we compare 3 replicas with 5 replicas, we get a better performance for 5 replicas. When we increase the number of users for Kubernetes, **due to excessive CPU and Memory usage, it's not able to cross across a specific thread count.**

13) Microservice – History - Direct

Microservice – History Service

Language - Go lang

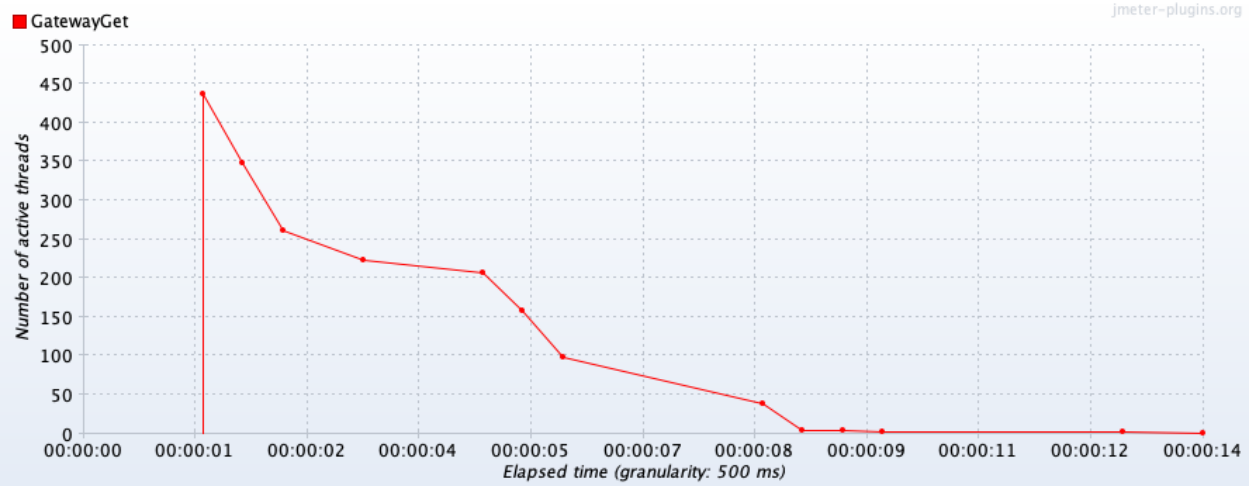
URL - /logging

Max Number of users: 650

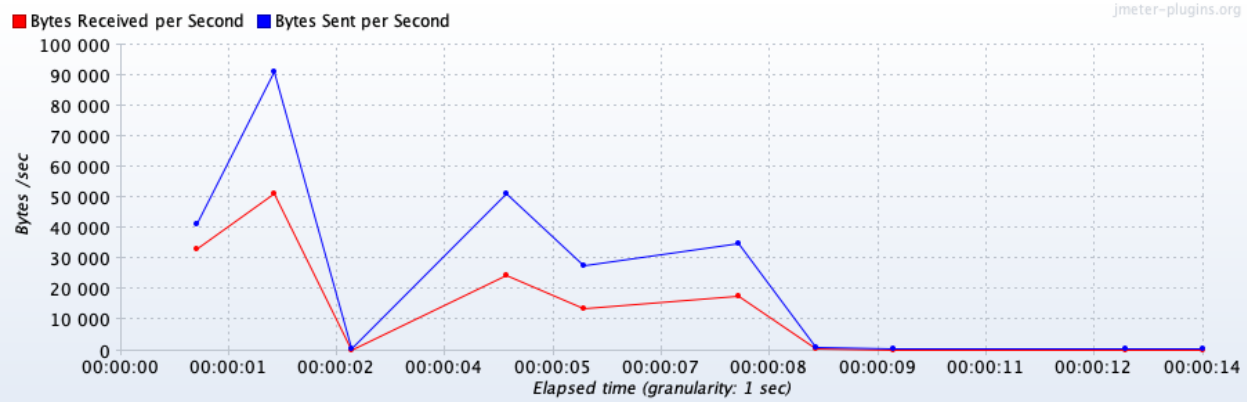
Error rate: 10.92%

Throughput: 384.8/secs

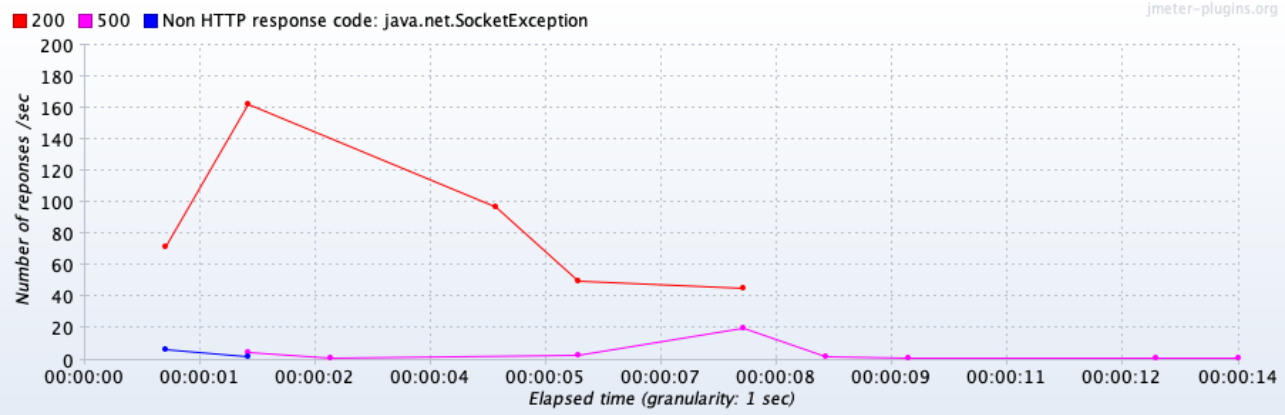
Active Threads



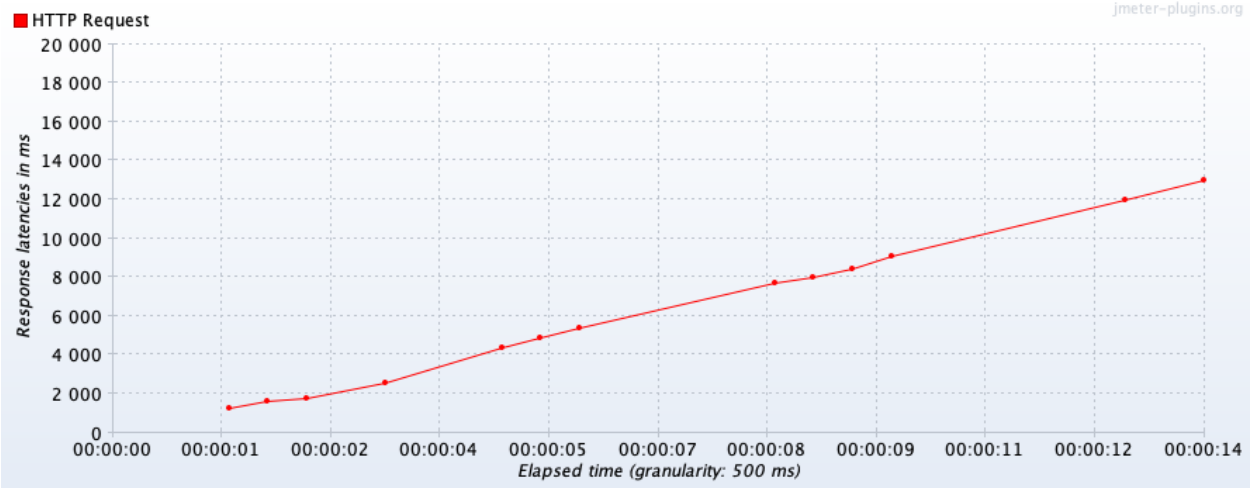
Bytes Throughput



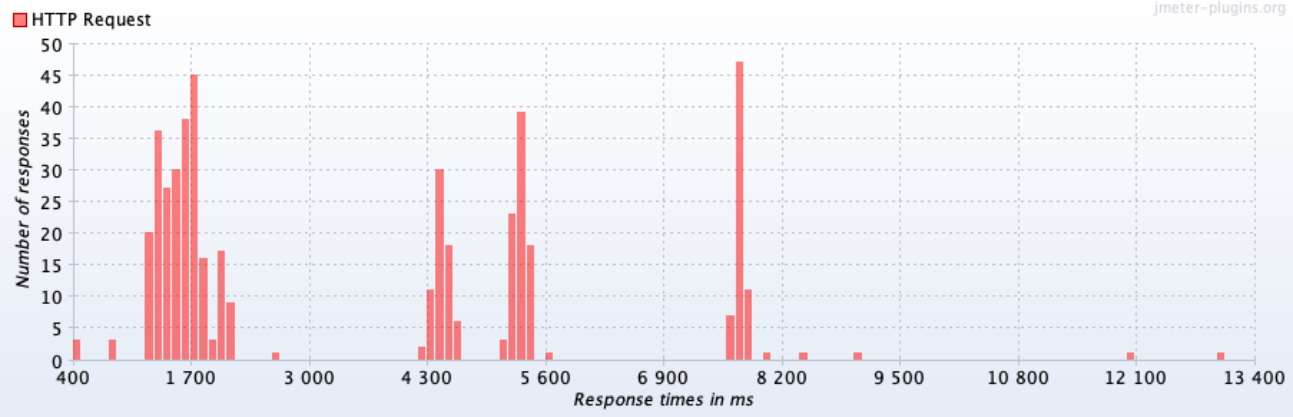
Response Codes Per Second



Response Latencies Over Time



Response Times Distribution



14**) Microservice – History - Gateway**

Language - Springboot/go

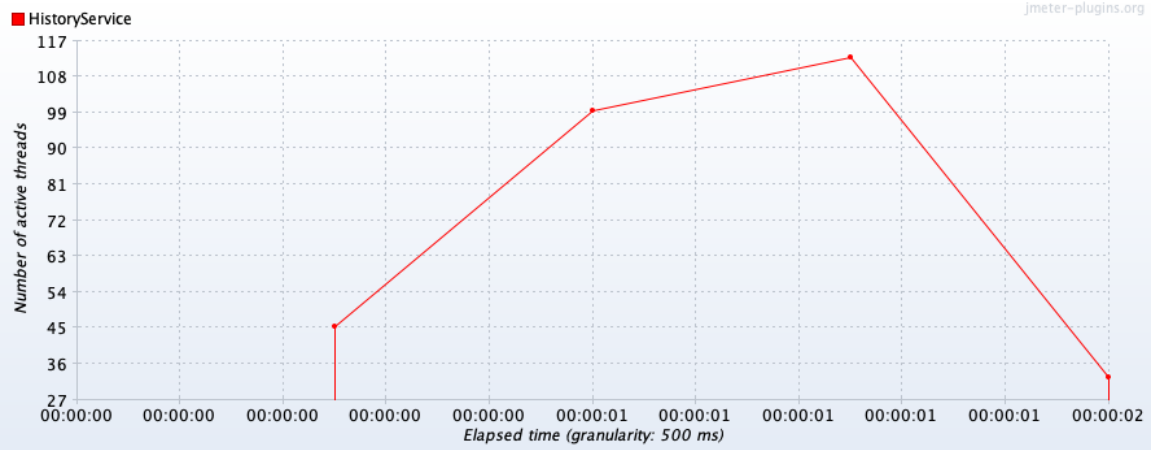
URL - /history-service/api/v1/logs

Max Number of users: 475

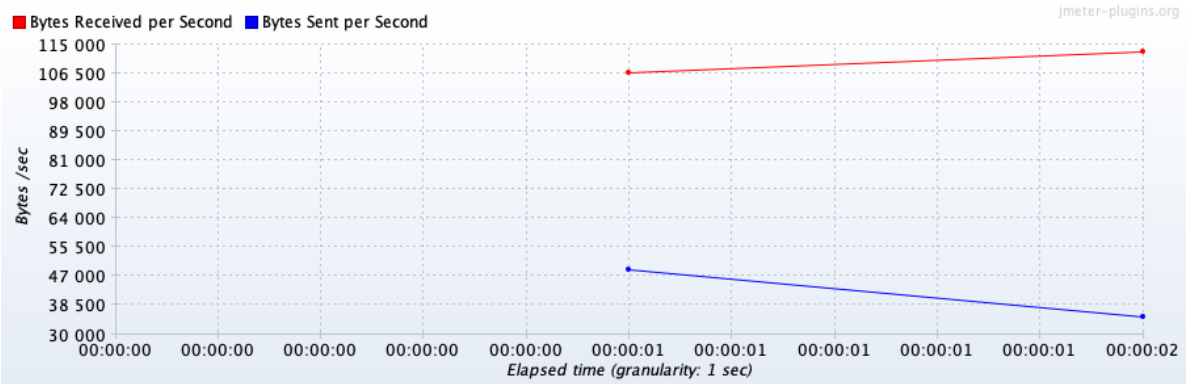
Error rate: 9.26%

Throughput: 34.1/sec

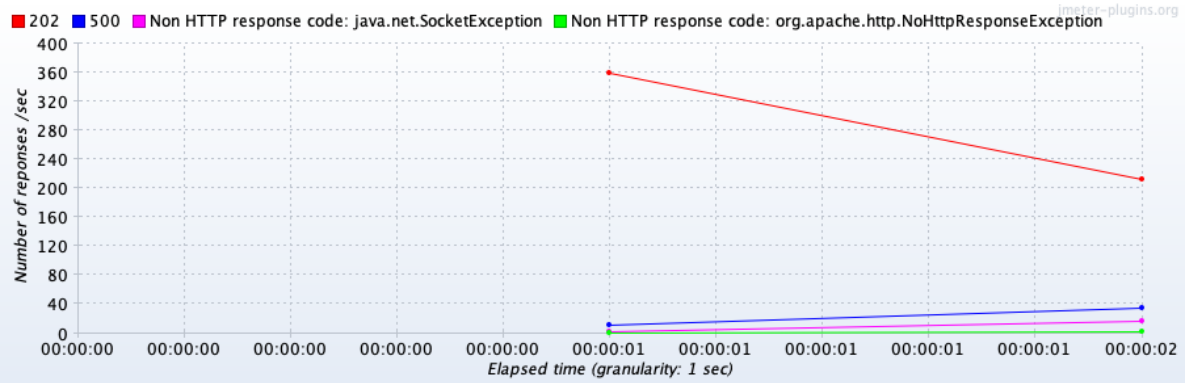
Active Threads



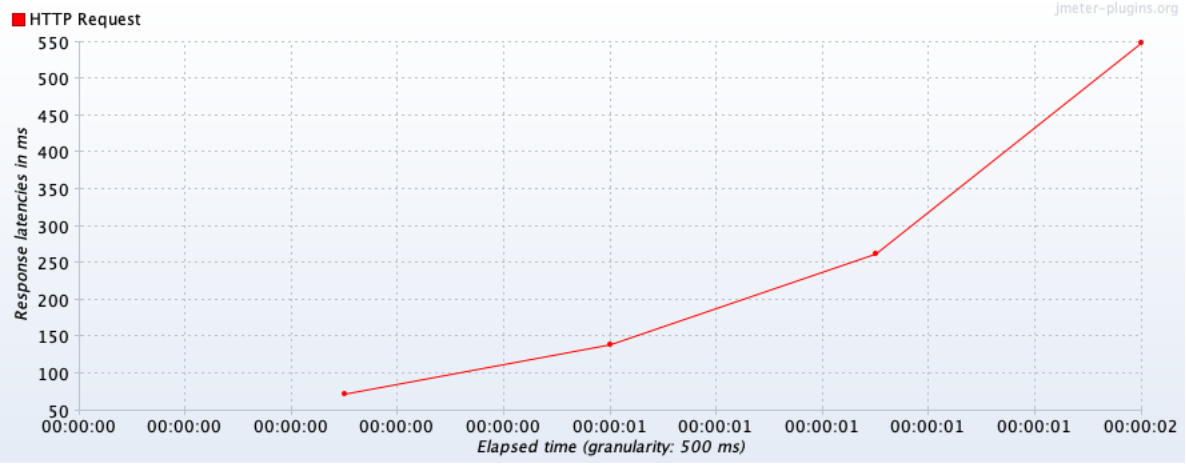
Bytes Throughput



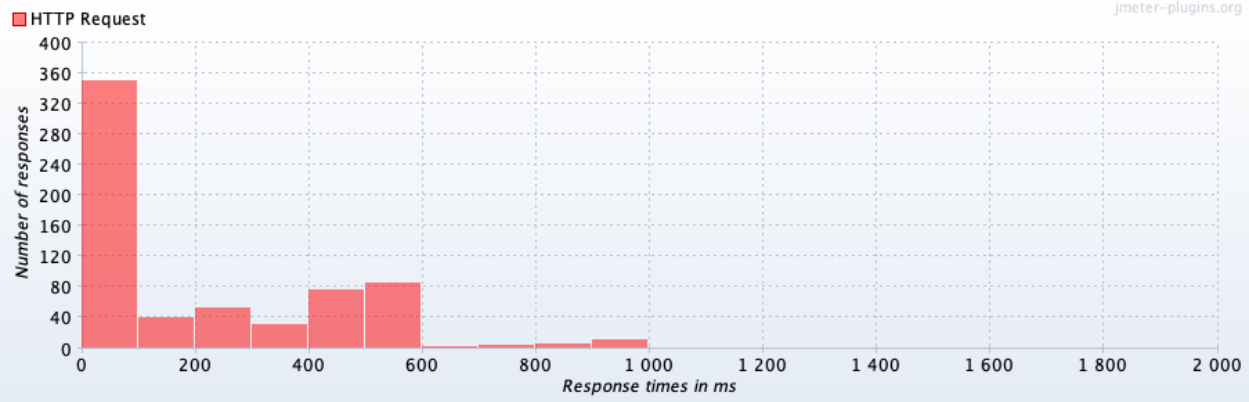
Response Codes Per Second



Response Latencies Over Time



Response Times Distribution



Comparison between History – Direct, History– Gateway

Name	Max Users	Error Rate	Throughput (Max)
History – Direct	650	11%	384.8/sec
History - Gateway	475	9.26%	34.1/sec

15) Microservice – History – GET Request

Language - Go lang

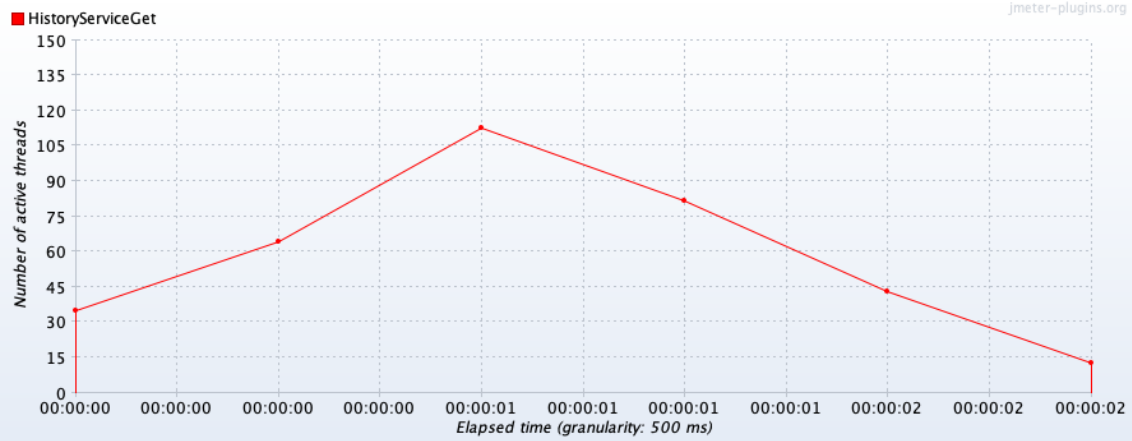
URL - /history-service/api/v1/user-history/1

Max Number of users: 200

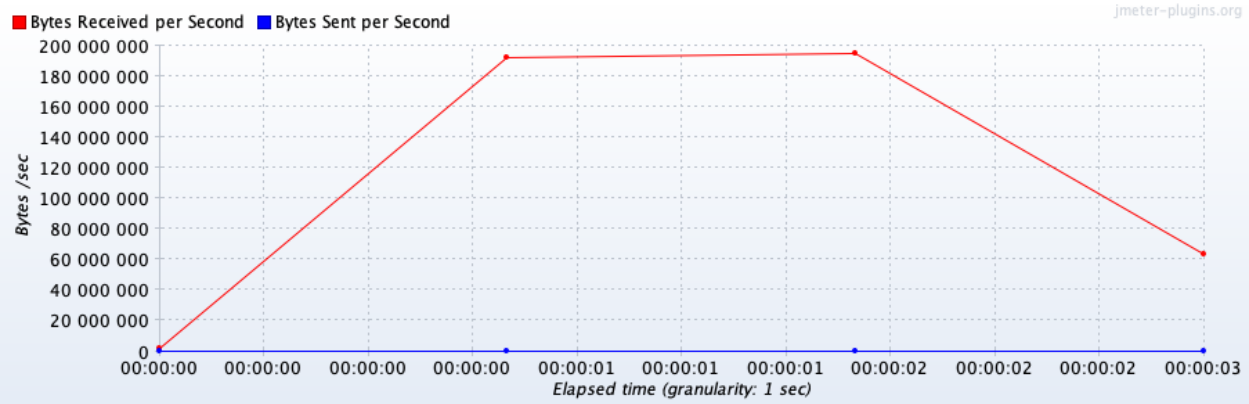
Error rate: 11.50%

Throughput: 79.8/sec

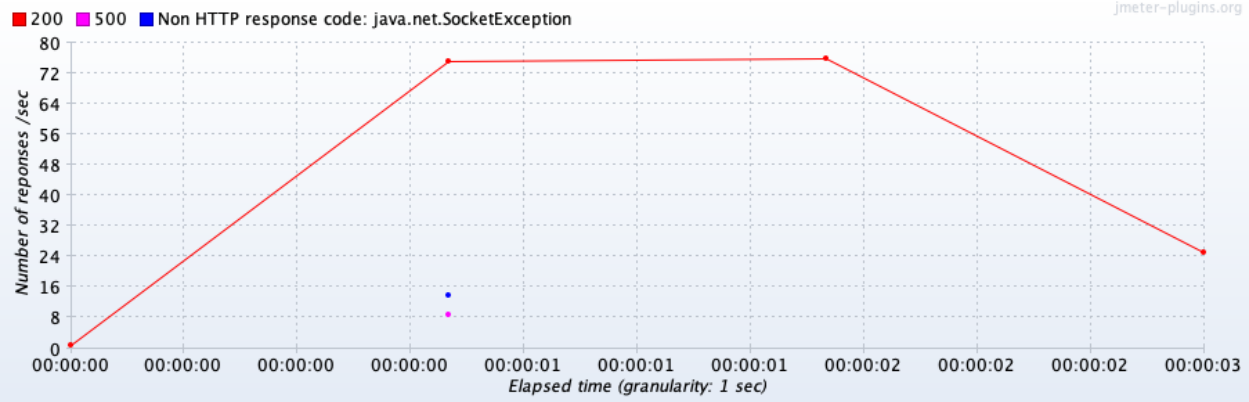
Active Threads



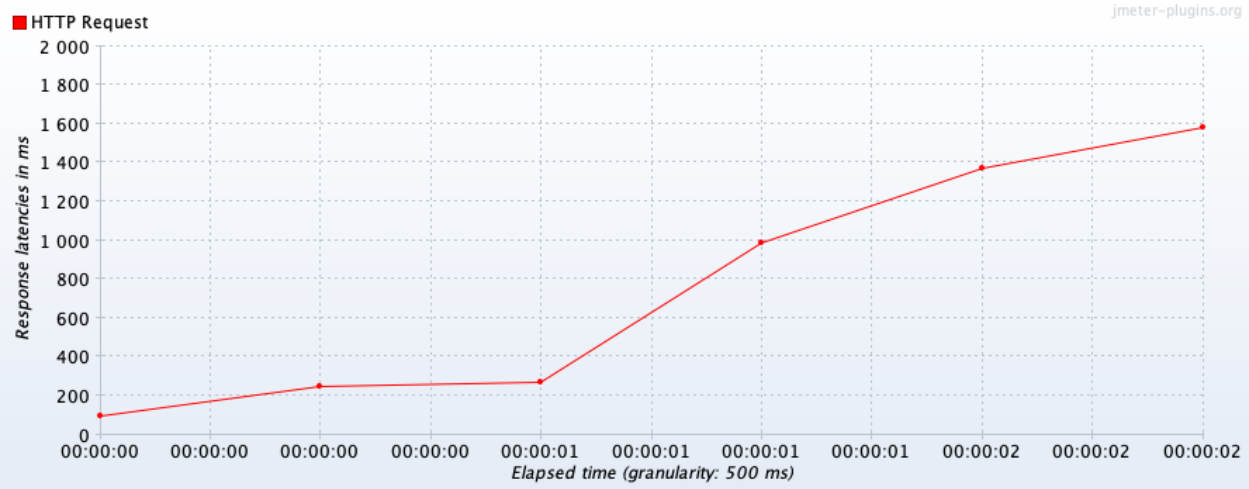
Bytes Throughput



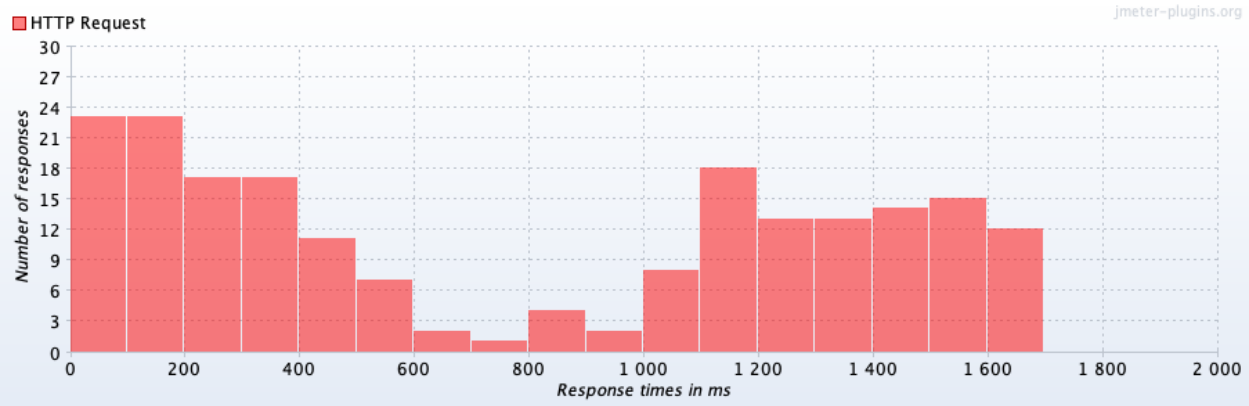
Response Codes Per Second



Response Latencies Over Time



Response Times Distribution



16) History - POST request - Kubernetes(3 Replicas - D)

Microservice – History

Language - go lang

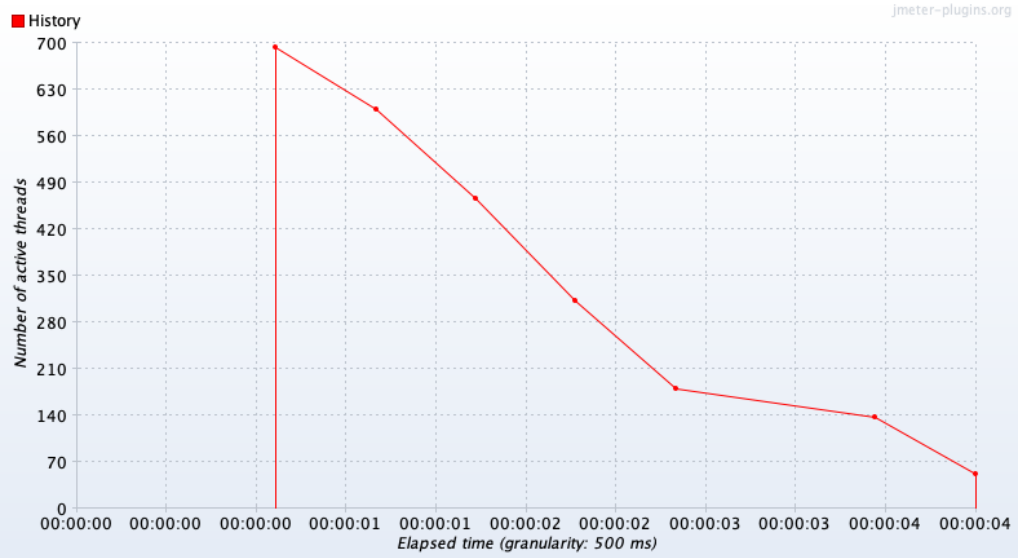
URL -/history-service/api/v1/logs

Max Number of users: 700

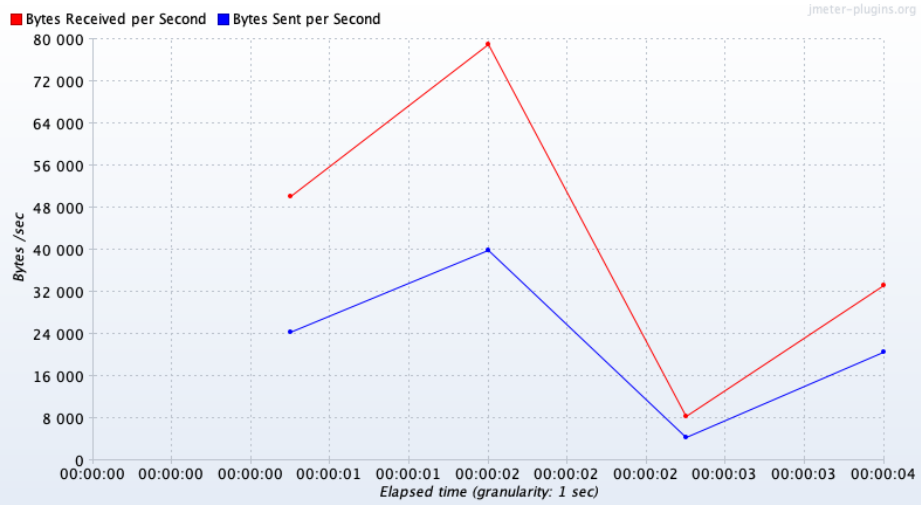
Error rate: 8.57%

Throughput: 161.6/sec

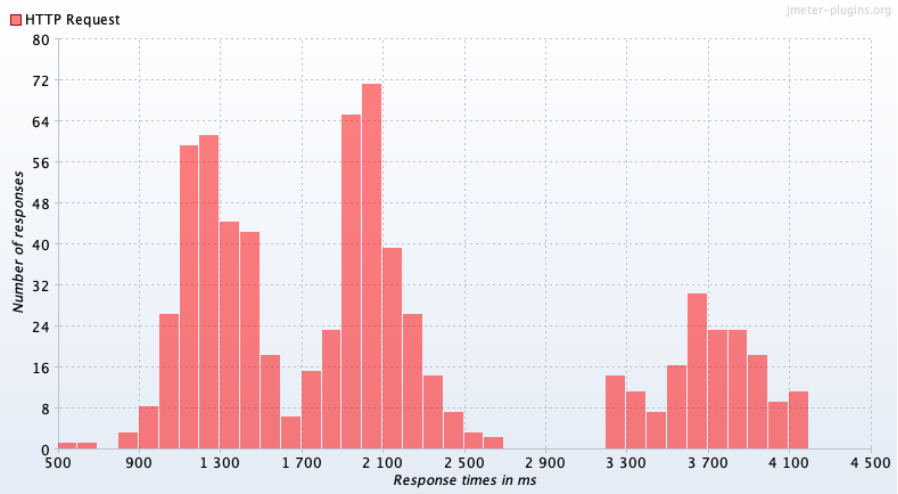
Active Threads



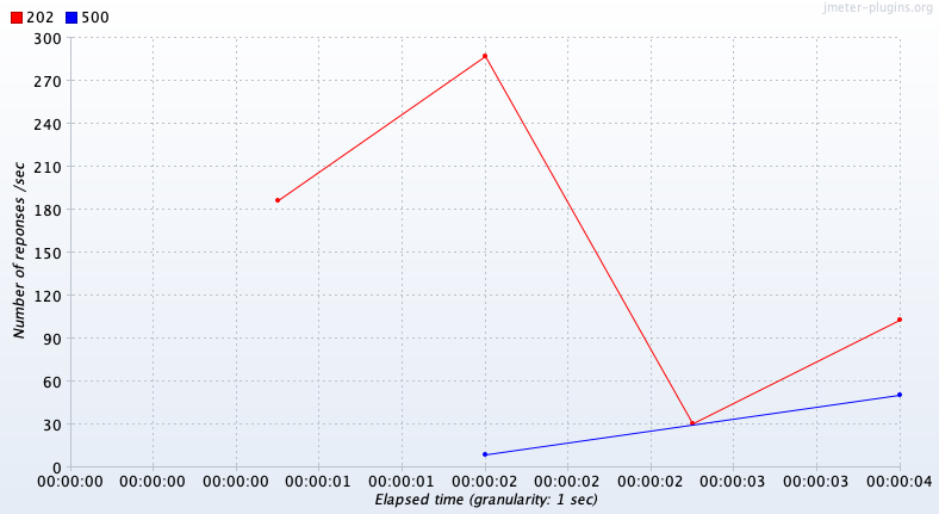
Bytes Throughput



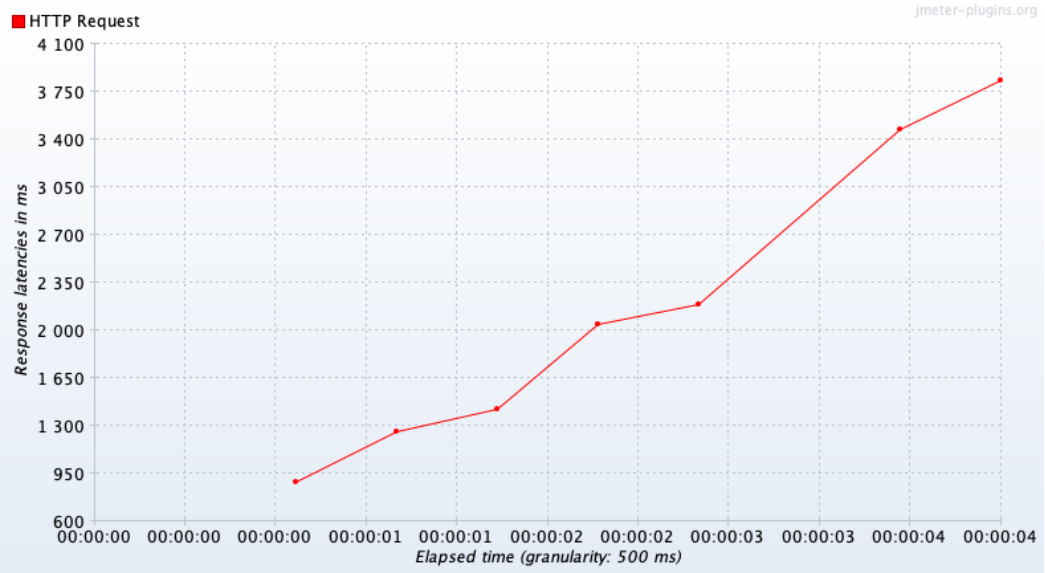
Response Times Distribution



Response Codes Per Second



Response Latencies Over Time



17) History -POST request - Kubernetes(5 Replicas - D)

Microservice – History

Language - go lang

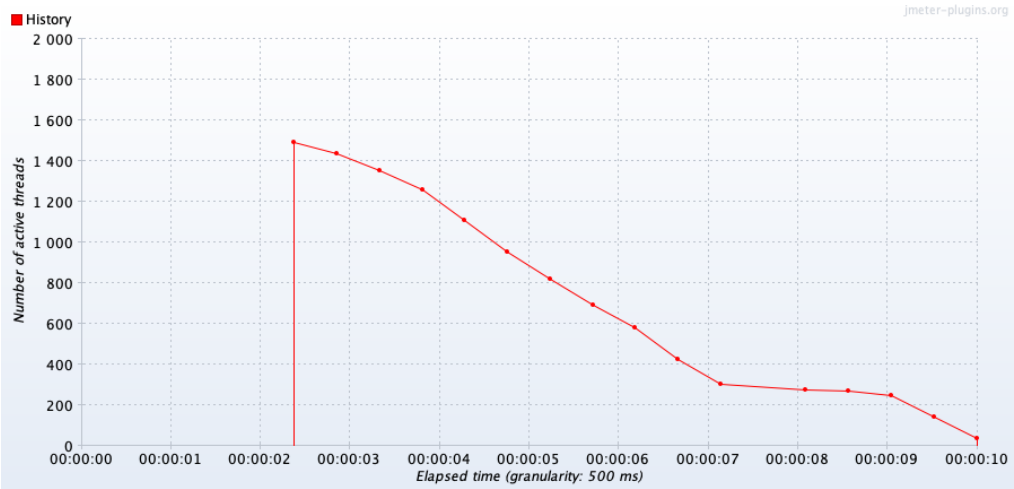
URL -/history-service/api/v1/logs

Max Number of users: 1500

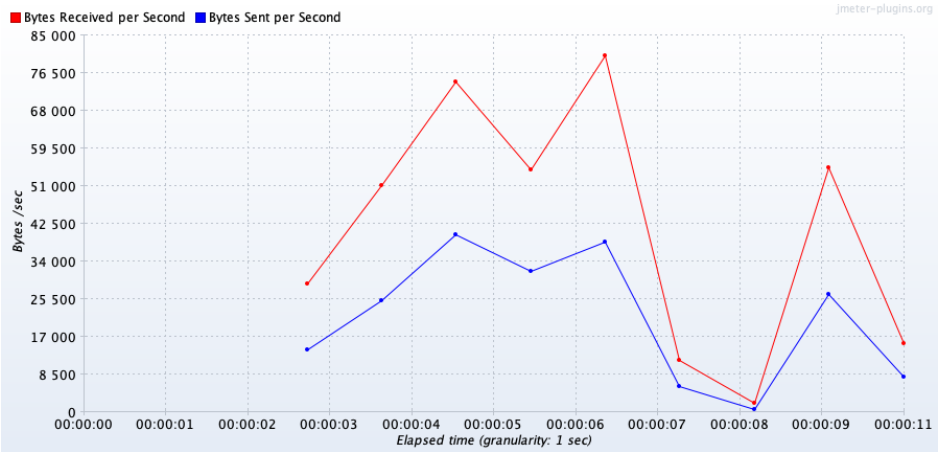
Error rate: 7.73%

Throughput: 149.6/sec

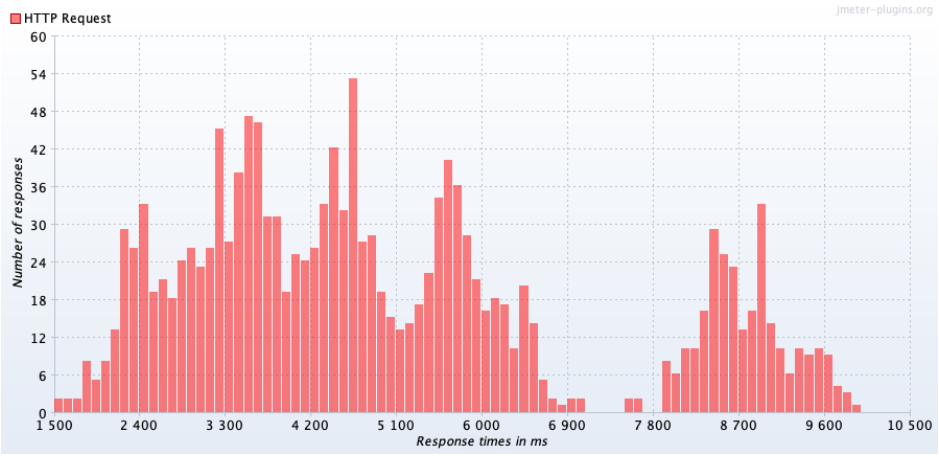
Active Threads



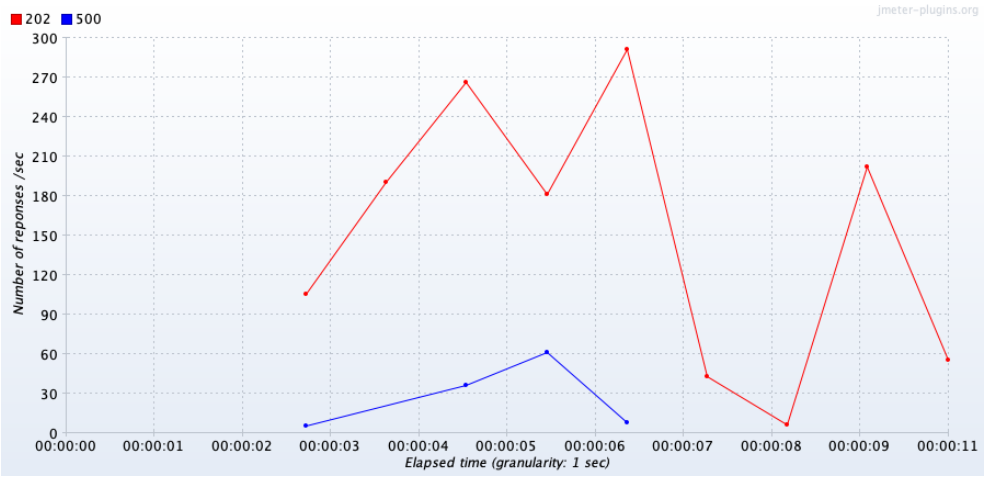
Bytes Throughput



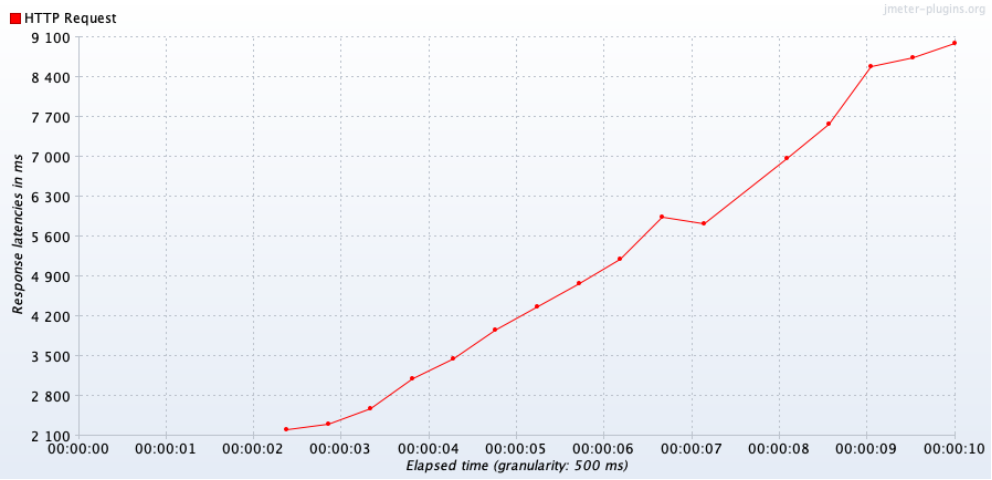
Response Times Distribution



Response Codes Per Second



Response Latencies Over Time



18) History -GET request - Kubernetes(3 Replicas - D)

Microservice – History

Language - go lang

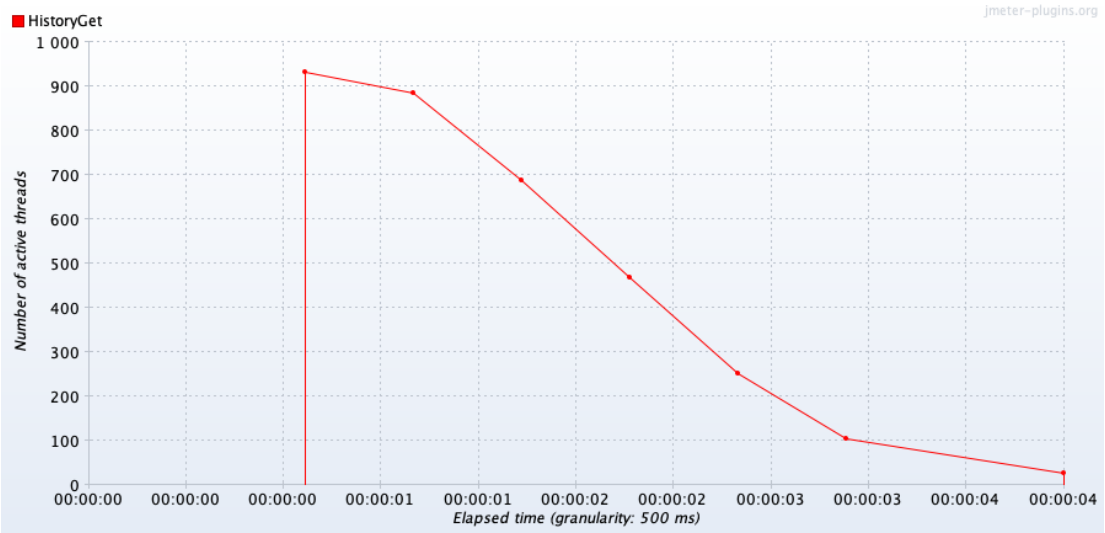
URL -/history-service/api/v1/user-history/1

Max Number of users: 1000

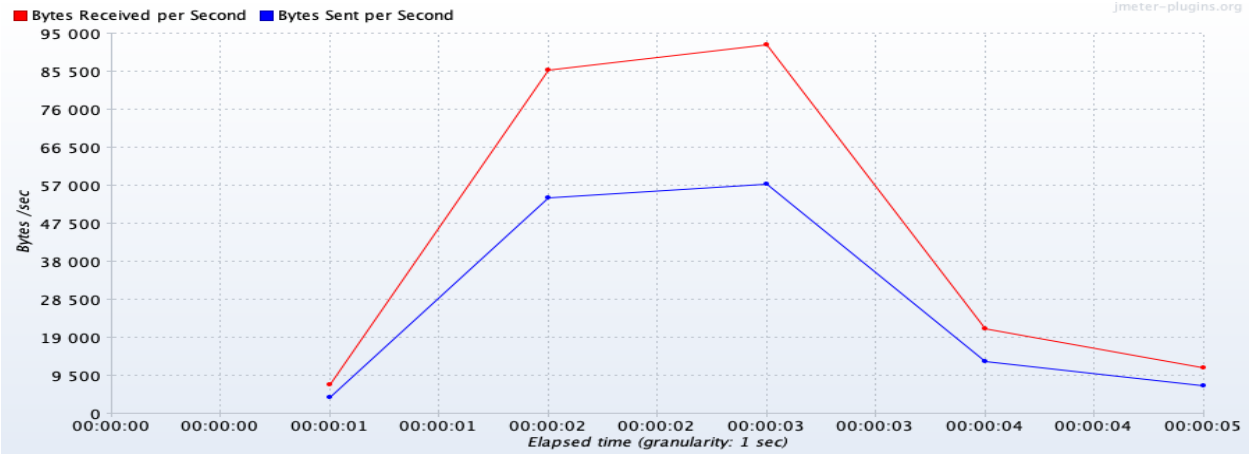
Error rate: 6.0%

Throughput: 240.4/sec

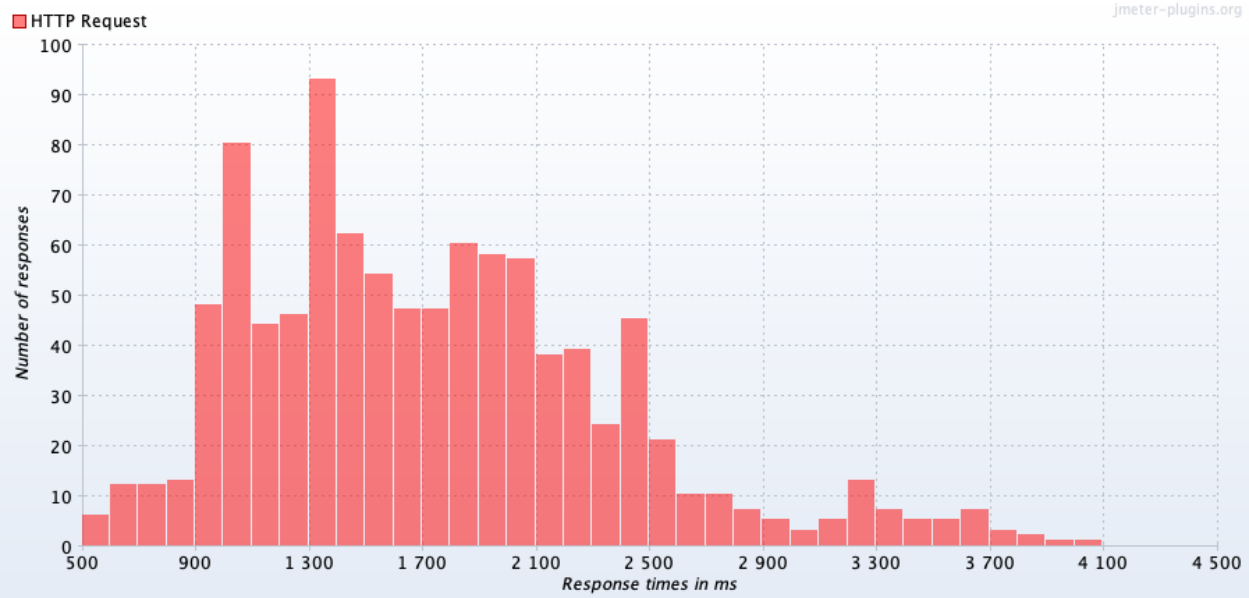
Active Threads



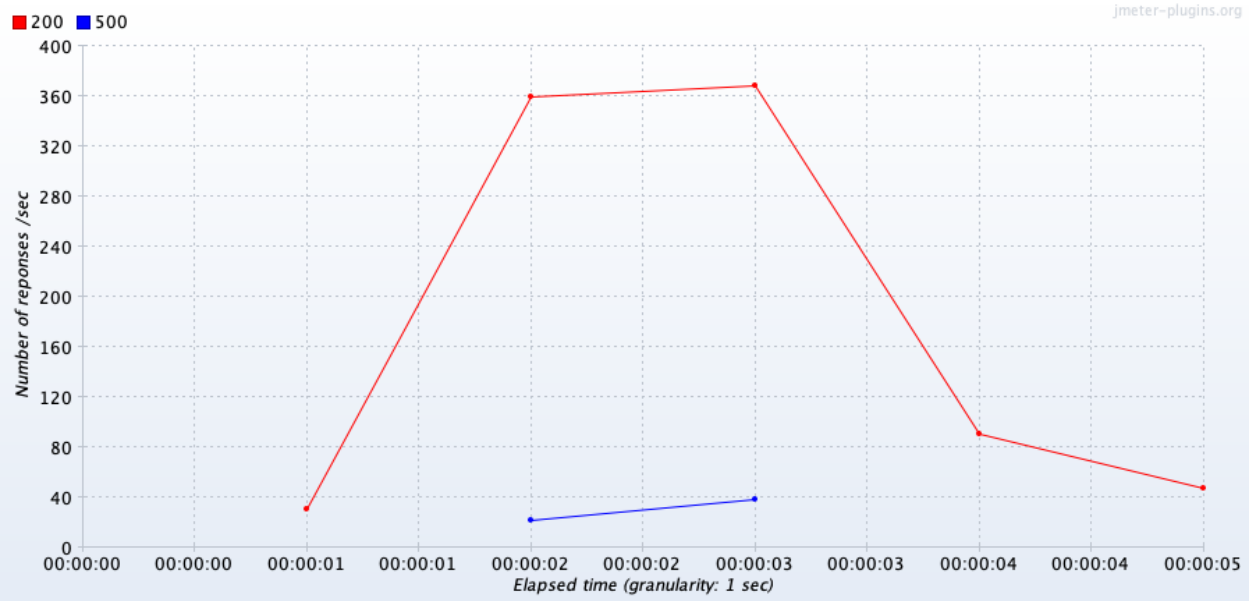
Bytes Throughput



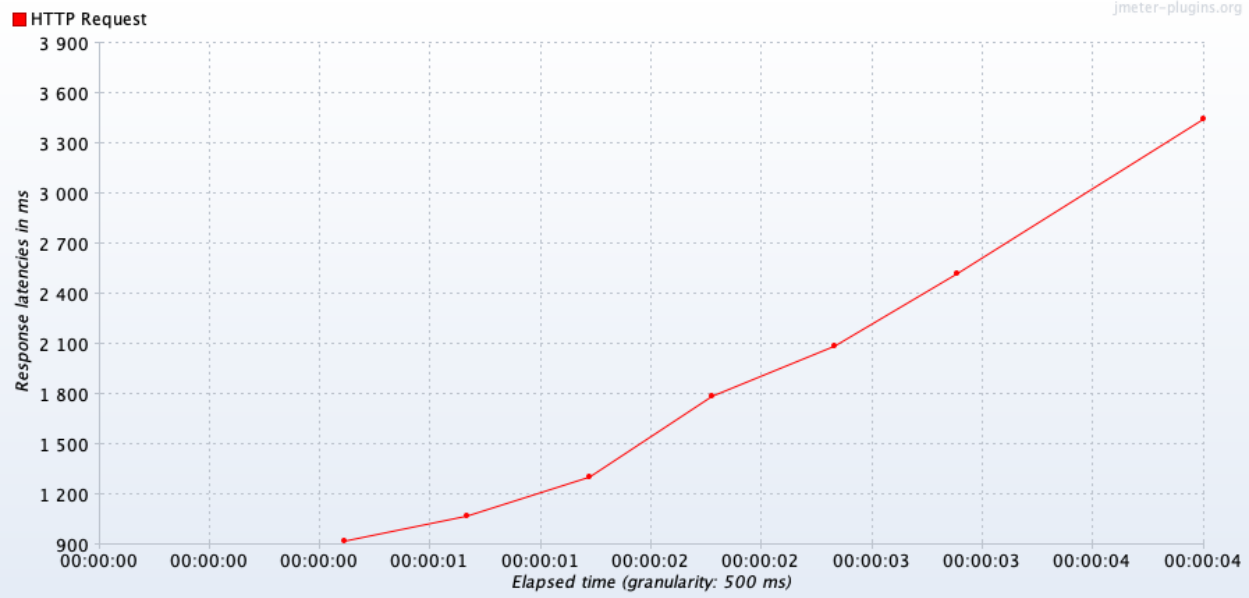
Response Times Distribution



Response Codes Per Second



Response Latencies Over Time



19) History -GET request - Kubernetes(5 Replicas - D)

Microservice – History

Language - go lang

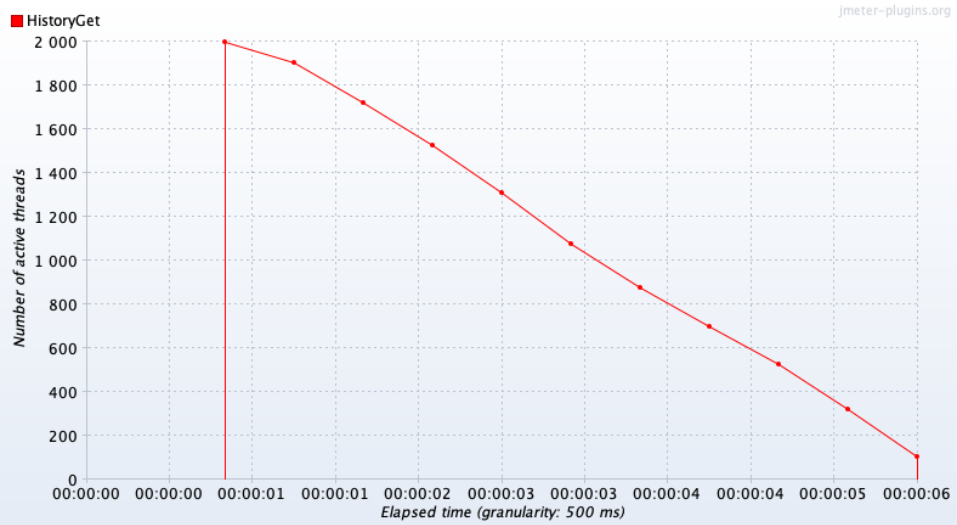
URL -/history-service/api/v1/user-history/1

Max Number of users: 2000

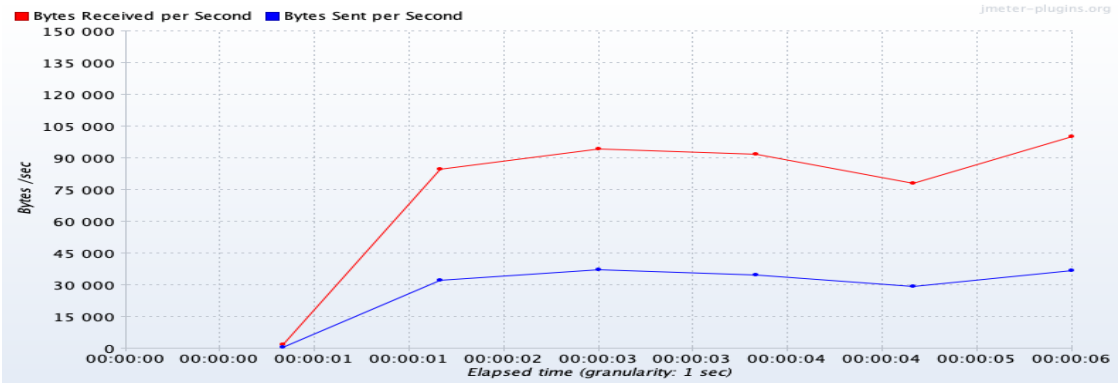
Error rate: 3.10%

Throughput: 334.4/sec

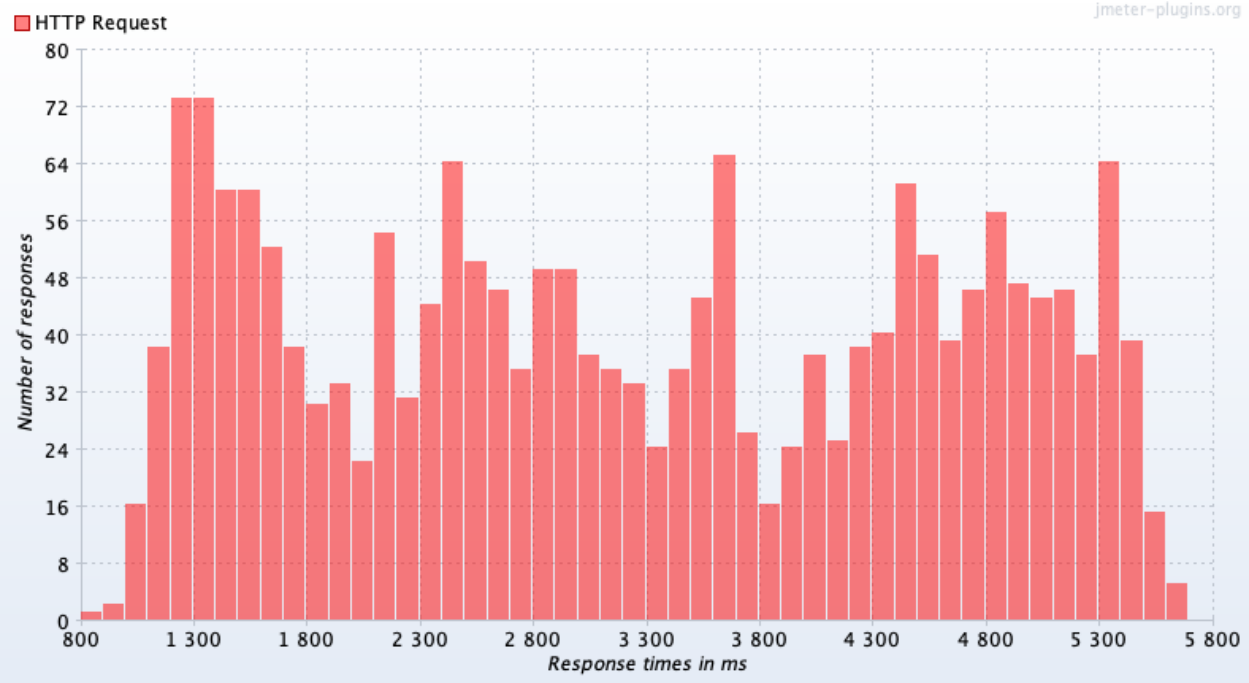
Active Threads



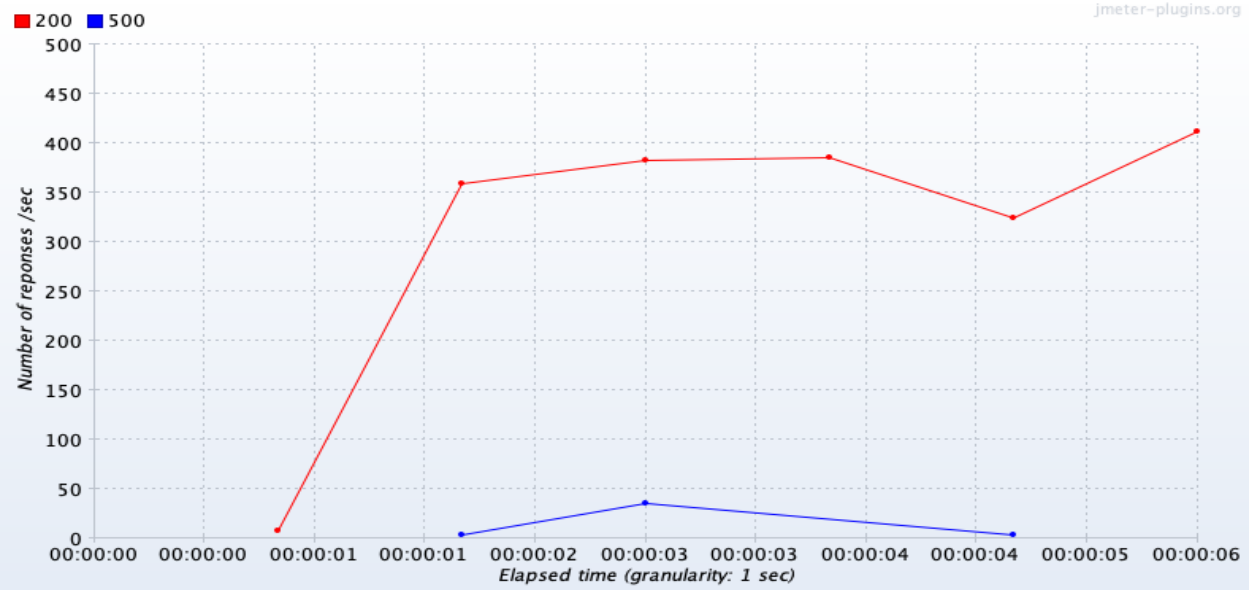
Bytes Throughput



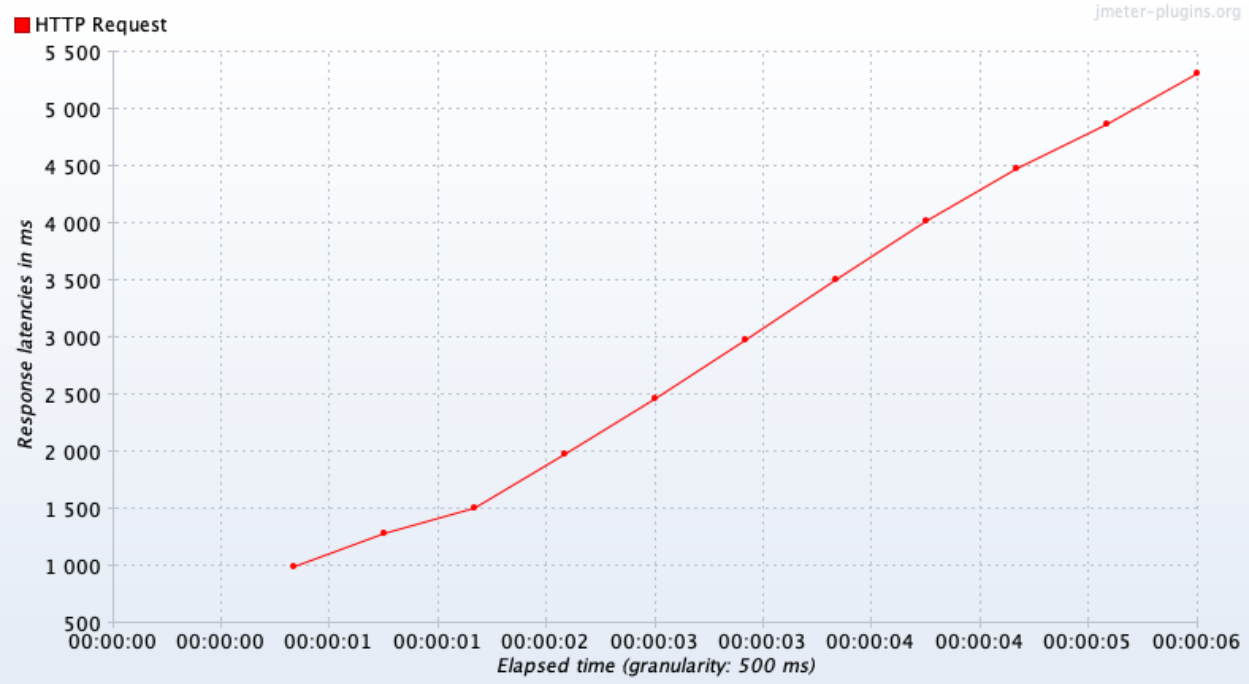
Response Times Distribution



Response Codes Per Second



Response Latencies Over Time



History service overview:

Name	Max users	Error rate	Throughput
History-POST – Direct – 1 replica	255	10.92%	384.8/sec
History-POST – Direct – 3 repli-cas	700	8.57%	161.6/sec
History - POST – 5 replicas	1500	7.73	149.1/sec
History - GET – 1 replicas	400	7.0%	258.1/sec
History - GET – 3 replicas	1000	6.0%	240.4/sec
History - GET – 5 replicas	2000	3.10	334.4/sec

We observed significant change in the number of users handled by history service when the number of replicas were changed from 1 to 3 and then to 5. The differences are shown in the table above.

Understanding Load Testing

The objective in load testing is to check the loads for every endpoint in which we start seeing significant failure rates. For this, we need to set a certain error threshold which we would deem to be satisfactory to deploy our application. After much deliberation and repeated testing on Jmeter, we decided to consider error rate of 10% as acceptable for that particular endpoint. As is shown earlier, we see the load handling capacity varying according to the method we use to reach that endpoint. In general, the microservices are able to handle and process requests for higher number of users when they are reached directly at that endpoint as opposed to when getting connected through the gateway. Below, we have shown a consolidated table which gives a better view of the dissimilarities for the endpoints. For most of the microservices such as Login, SignUp and History, while using Kubernetes, we see substantial improvement in application performance to withstand high loads due to the parallel replicas created within it. However, for plotting, as explained earlier, we don't see radical change in performance because of the memory leakage issue and extremely high CPU and memory utilization which causes our system to break down. For throughput, we observe that it remains similar or decreases as we use Kubernetes but that's because the number of threads are way larger when we use the replicas which means more processing load for the application.

One additional thing to note is that instead of using a ramp-up period, we are **sending all the requests at once.**

Microservice	Type	Endpoint	Number of users	Error	Throughput
Registry	Login - Direct	/registry/api/v1/user/login	580	9%	263.3/sec
Registry	Login - Gateway	/login	430	11%	254.6/sec
Registry	Login - Direct(3 replica)	/registry/api/v1/user/login	1998	8.7%	400.7/sec
Registry	Login - Direct(5 replica)	/registry/api/v1/user/login	1997	5.61%	269.8/sec
Registry	SignUp - Direct	/registry/api/v1/user/signUp	560	8.39%	279.3/sec
Registry	SignUp - Gateway	/signUp	450	13.11%	274.4/sec
Registry	SignUp - Direct(3 replica)	/registry/api/v1/user/signUp	1920	10.78%	438.5/sec
Registry	SignUp - Direct(5 replica)	/registry/api/v1/user/signUp	2220	12.48%	395.0/sec
Plotting	Direct	/getPlottedData	45	0%	12.5/min
Plotting	Gateway	/plotting	36	28%	15.8/min
Plotting	Kubernetes(1 replica)	/getPlottedData	20	0	18.6/min

Plotting	Kubernetes(3 replica)	/getPlottedData	40	0	32.8/min
Plotting	Kubernetes(5 replica)	/getPlottedData	50	0	47.9/min
History	Direct	/history-service/api/v1/logs	650	11%	384.8/sec
History	Gateway	/history-service/api/v1/logs	475	9.26%	34.1/sec
History	Direct(1 replica)	/history-service/api/v1/logs	255	10.92%	384.8/sec
History	Direct(3 replica)	/history-service/api/v1/logs	700	8.57%	161.6/sec
History	Direct(5 replica)	/history-service/api/v1/logs	1500	7.73	149.1/sec
History	Direct(1 replica)	/history-service/api/v1/user-history/1	400	7.0%	258.1/sec
History	Direct(3 replica)	/history-service/api/v1/user-history/1	1000	6.0%	240.4/sec
History	Direct(5 replica)	/history-service/api/v1/user-history/1	2000	3.10%	344.4/sec

SPIKE TESTING

The table above shows the number of users handled by each service before showing increase in the error rate. We increased the number of users gradually (example: 100, 200, 500 …) and observed the error rate. When system encounters sudden increase in the number of users it shows high error rate. In case of sudden increase in the number of users, we observed sharp increase in the error rate.

Apart from error rate, we observed if the service has multiple replicas in place, spike in the number of users caused the automatic restart of replicas.

The spike in the number of user and the error rate also depends upon the number of existing replicas. If the only one service is handling the all the incoming request with no replica, the error spike observed with less number of users. More the number of replica, less are the chances to get high error.

We can conclude that, if we implement auto scaling the system will be much reliable and will show more fault tolerance.

Microservice	Type	Endpoint	Number of users	Error	Throughput
Registry	Login - Direct	/registry/api/v1/user/login	580	9%	263.3/sec
Registry	Login - Gateway	/login	430	11%	254.6/sec
Registry	Login - Direct(3 replica)	/registry/api/v1/user/login	1998	8.7%	400.7/sec
Registry	Login - Direct(5 replica)	/registry/api/v1/user/login	1997	5.61%	269.8/sec
Registry	SignUp - Direct	/registry/api/v1/user/signUp	560	8.39%	279.3/sec
Registry	SignUp - Gateway	/signUp	450	13.11%	274.4/sec
Registry	SignUp - Direct(3 replica)	/registry/api/v1/user/signUp	1920	10.78%	438.5/sec
Registry	SignUp - Direct(5 replica)	/registry/api/v1/user/signUp	2220	12.48%	395.0/sec
Plotting	Direct	/getPlottedData	45	0%	12.5/min

Plotting	Gateway	/plotting	36	28%	15.8/min
Plotting	Kubernetes(1 replica)	/getPlottedData	20	0	18.6/min
Plotting	Kubernetes(3 replica)	/getPlottedData	40	0	32.8/min
Plotting	Kubernetes(5 replica)	/getPlottedData	50	0	47.9/min
History	Direct	/history-service/api/v1/logs	650	11%	384.8/sec
History	Gateway	/history-service/api/v1/logs	475	9.26%	34.1/sec
History	Direct(1 replica)	/history-service/api/v1/logs	255	10.92%	384.8/sec
History	Direct(3 replica)	/history-service/api/v1/logs	700	8.57%	161.6/sec
History	Direct(5 replica)	/history-service/api/v1/logs	1500	7.73	149.1/sec
History	Direct(1 replica)	/history-service/api/v1/user-history/1	400	7.0%	258.1/sec
History	Direct(3 replica)	/history-service/api/v1/user-history/1	1000	6.0%	240.4/sec
History	Direct(5 replica)	/history-service/api/v1/user-history/1	2000	3.10%	344.4/sec

UNDERSTANDING FAULT TOLERANCE OF SYSTEM:

Sample :

- For testing, we used the Registry microservices Login Api, which verifies data in Mongodb and generates a jwt token, which is then placed into the database to keep the session going.
- For testing, we used Kubernetes 5 Replicas.

Process:

- Deleted one pod while requested threads were running in background.

Observations:

- There was no mistake made when one pod failed (1 replica).
- Kubernetes was still in charge of threads.
- Because of the limited pods, the error rate rose for the same number of threads.
- There was no discernible influence on throughput.
- We believe, that Kubernetes(Load Balancer Management) initially divides number of total threads among services, and when a service fails, the threads delegated to it fails.

We concluded to these results, as we observed error rate increasing with increasing number of pod failures.

Comparison between Kubernetes Pod Failures:

Name	Number of Users	Error Rate	Throughput
0 Replica Failure	1500	0%	194.1/sec
1 Replicas-Failure	1500	9%	212.0/sec
3 Replicas-Failure	1500	55.27%	173.0/sec

Microservice – Dev Registry

Language - NodeJS

URL - /registry/api/v1/user/login

WHEN DELETING POD:

Max Number of users: 1500

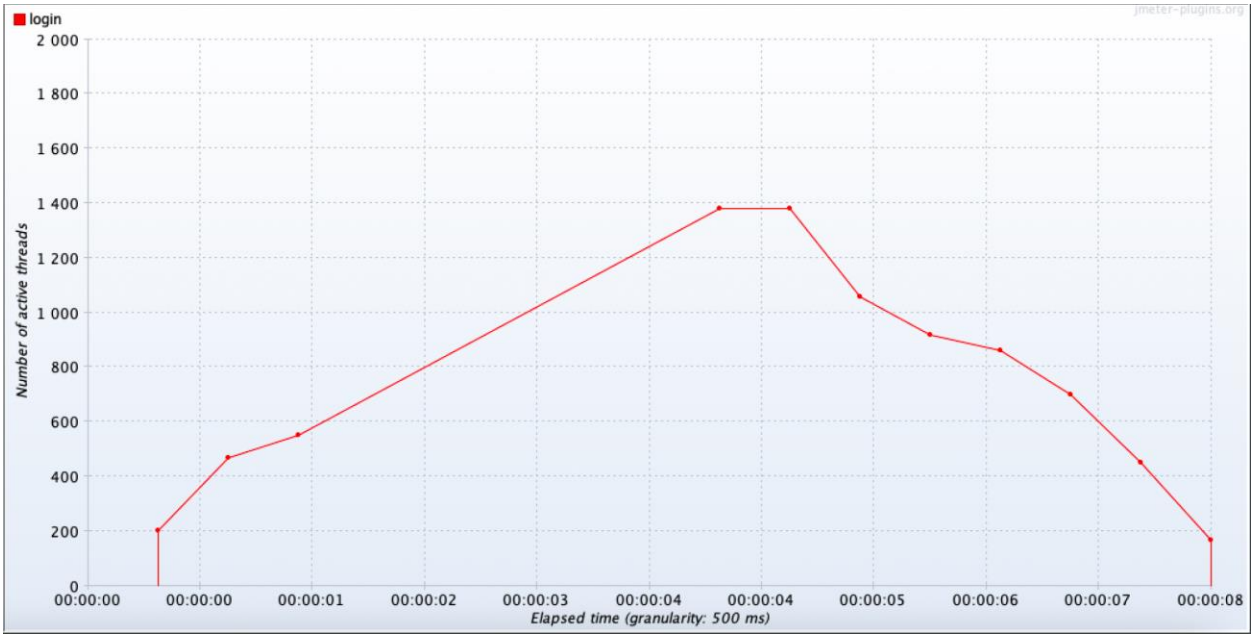
Error rate: 0%

Throughput: 194.1/sec

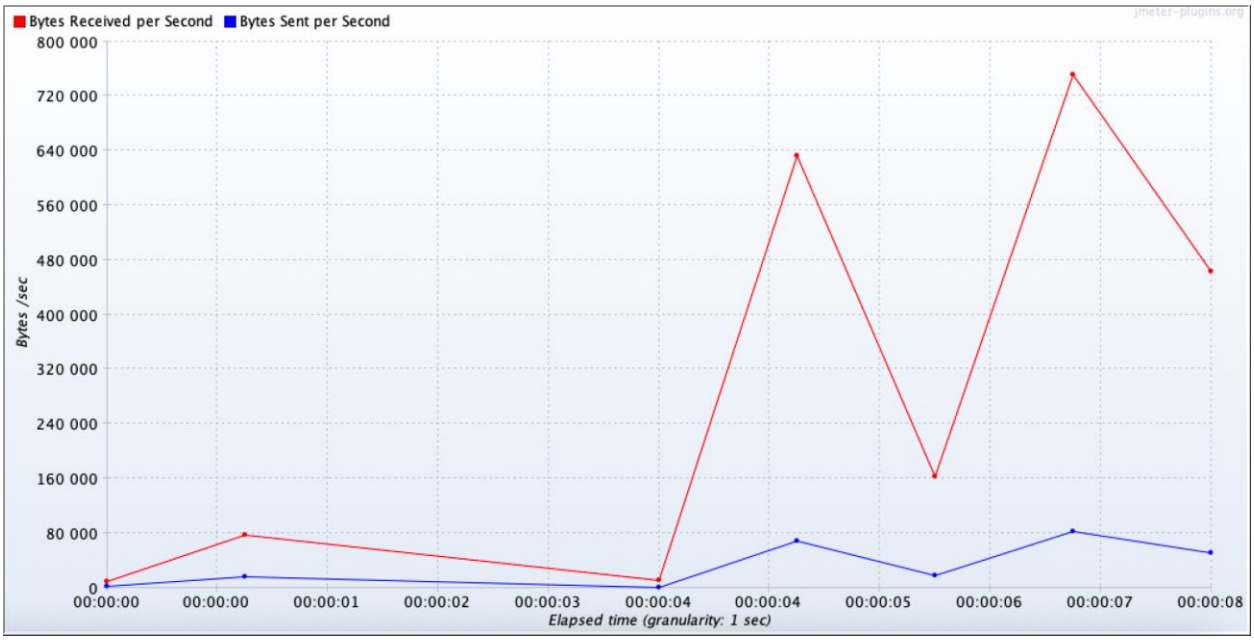
SUMMARY REPORT:

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received K...	Sent KB/sec
HTTP Requ...	1500	2271	2365	3500	3713	4238	121	4349	0.00%	194.1/sec	280.12	52.88
TOTAL	1500	2271	2365	3500	3713	4238	121	4349	0.00%	194.1/sec	280.12	52.88

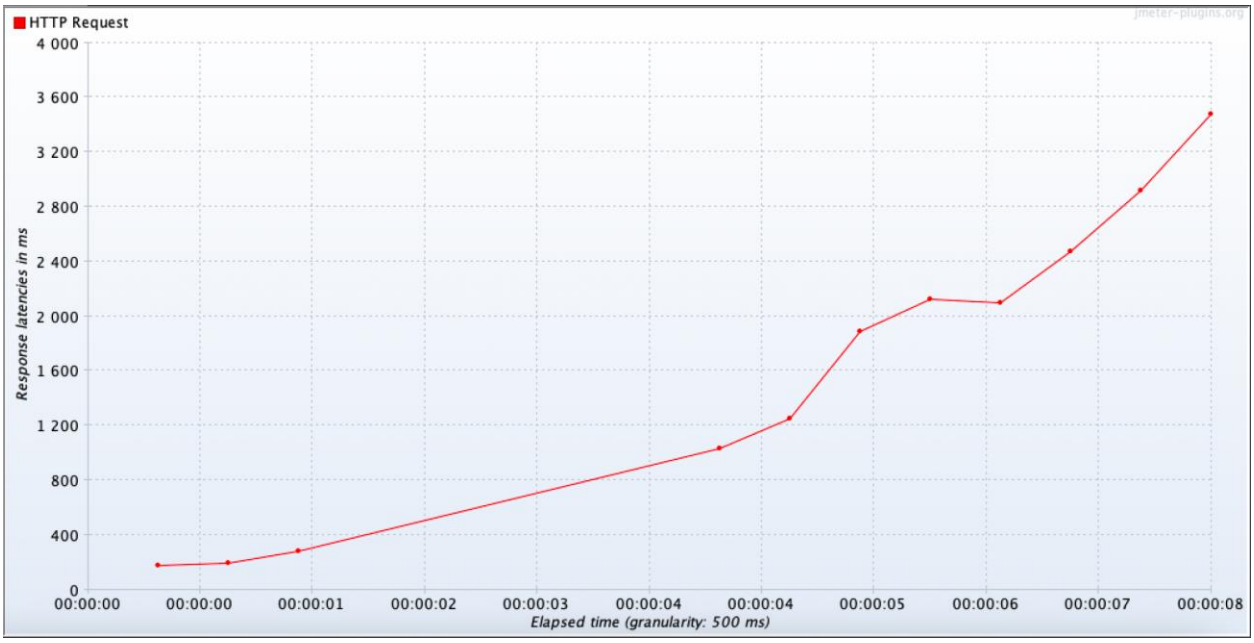
Active Threads



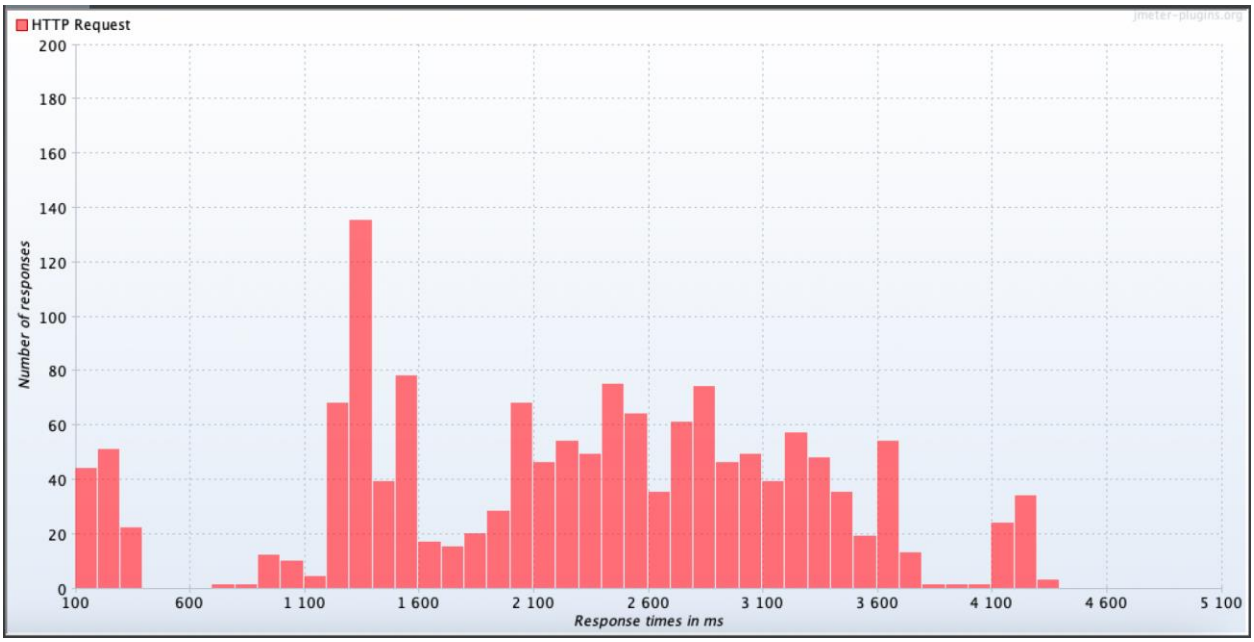
Bytes Throughput



Response Latencies Over Time



Response Times Distribution



WHEN DELETING 1 POD:

Deleted 1 pod while processes were requested in background.

Max Number of users: 1500

Error rate: 9.00%

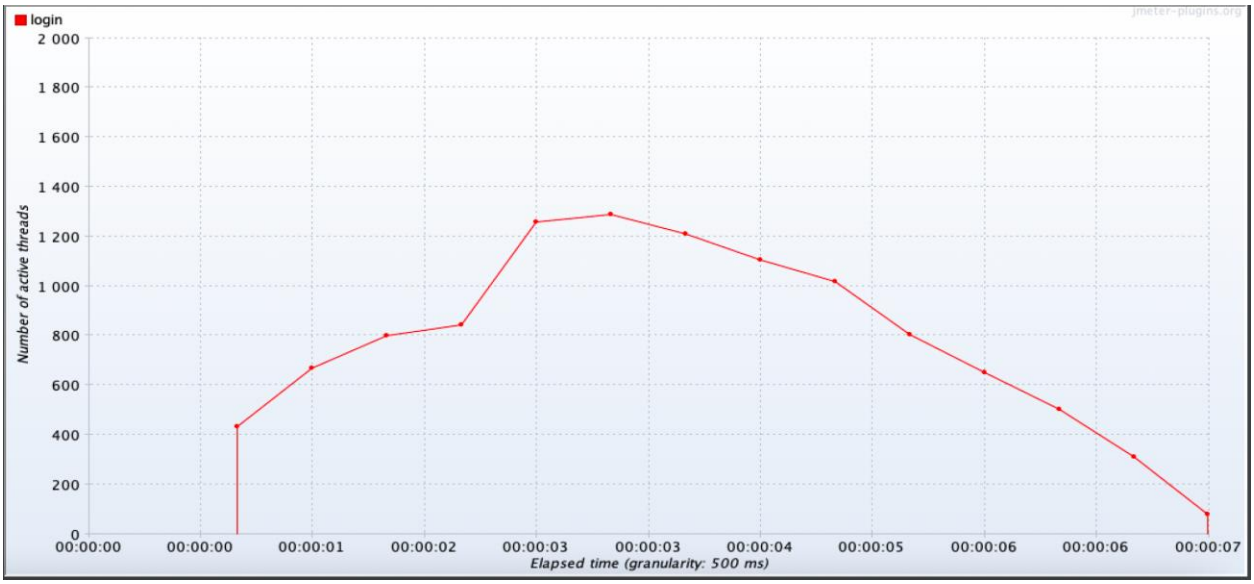
Throughput: 212.0/sec

```
(base) tanukansal@Tanus-MacBook-Pro Kub_Plotting_Service % kubectl get pods
NAME                                     READY   STATUS    RESTARTS      AGE
mongodb-deployment-97ff84f5b-ljrx7       1/1     Running   1 (18h ago)   2d15h
registry-service-deployment-858b6cc58f-ll8n9   1/1     Running   0             15h
registry-service-deployment-858b6cc58f-pj4w6   1/1     Running   0             15h
registry-service-deployment-858b6cc58f-rcsgl   1/1     Running   1 (18h ago)   2d12h
registry-service-deployment-858b6cc58f-t648m   1/1     Running   0             42s
registry-service-deployment-858b6cc58f-tsccm   1/1     Running   0             10h
(base) tanukansal@Tanus-MacBook-Pro Kub_Plotting_Service %  kubectl delete pod registry-service-deployment-858b6cc58f-ll8n9
pod "registry-service-deployment-858b6cc58f-ll8n9" deleted
(base) tanukansal@Tanus-MacBook-Pro Kub_Plotting_Service %
```

SUMMARY REPORT:

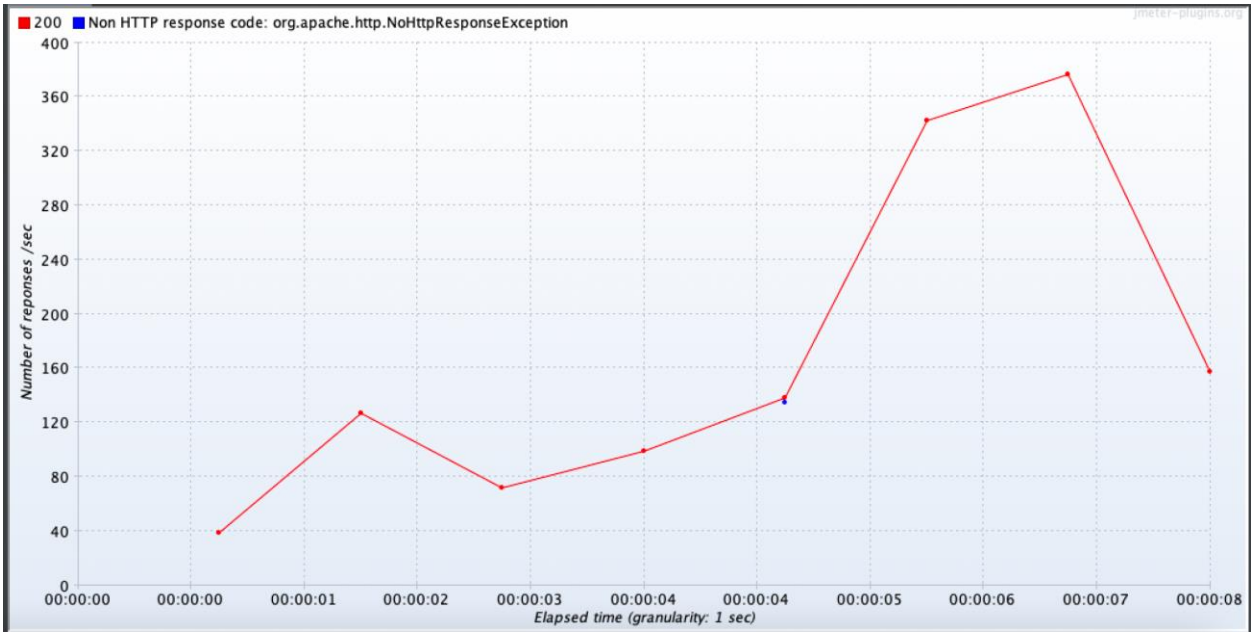
Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received K...	Sent KB/sec
HTTP Requ...	1500	2951	3080	4503	4664	4892	195	4985	9.00%	212.0/sec	316.71	52.55
TOTAL	1500	2951	3080	4503	4664	4892	195	4985	9.00%	212.0/sec	316.71	52.55

Active Threads

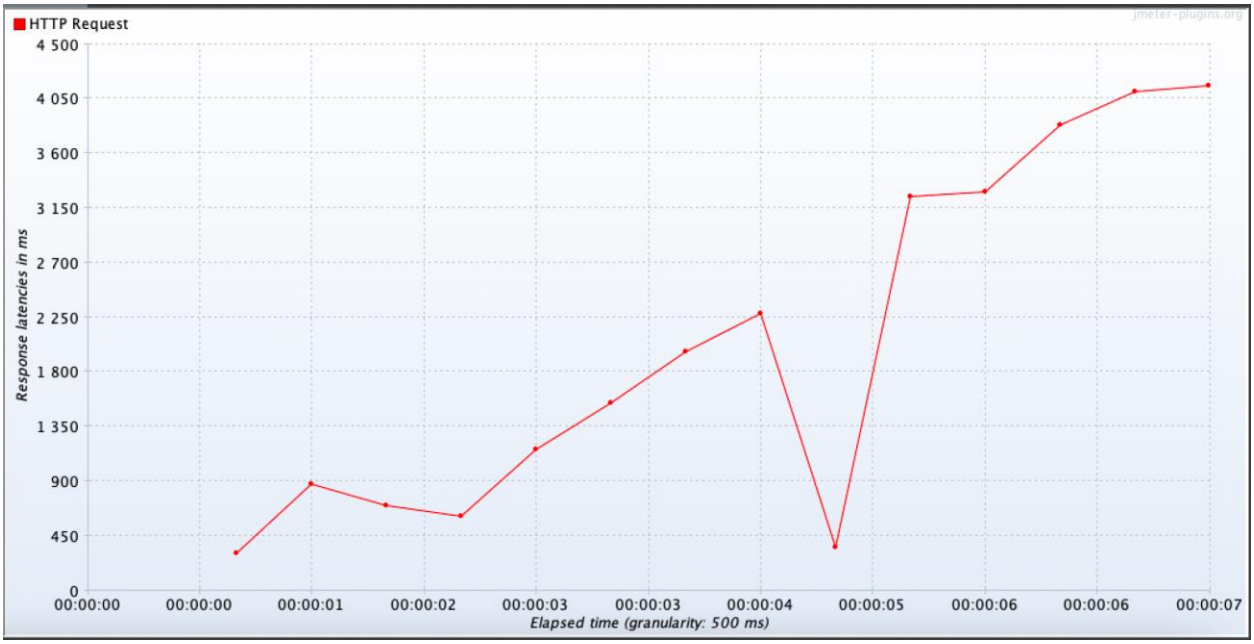


Bytes Throughput

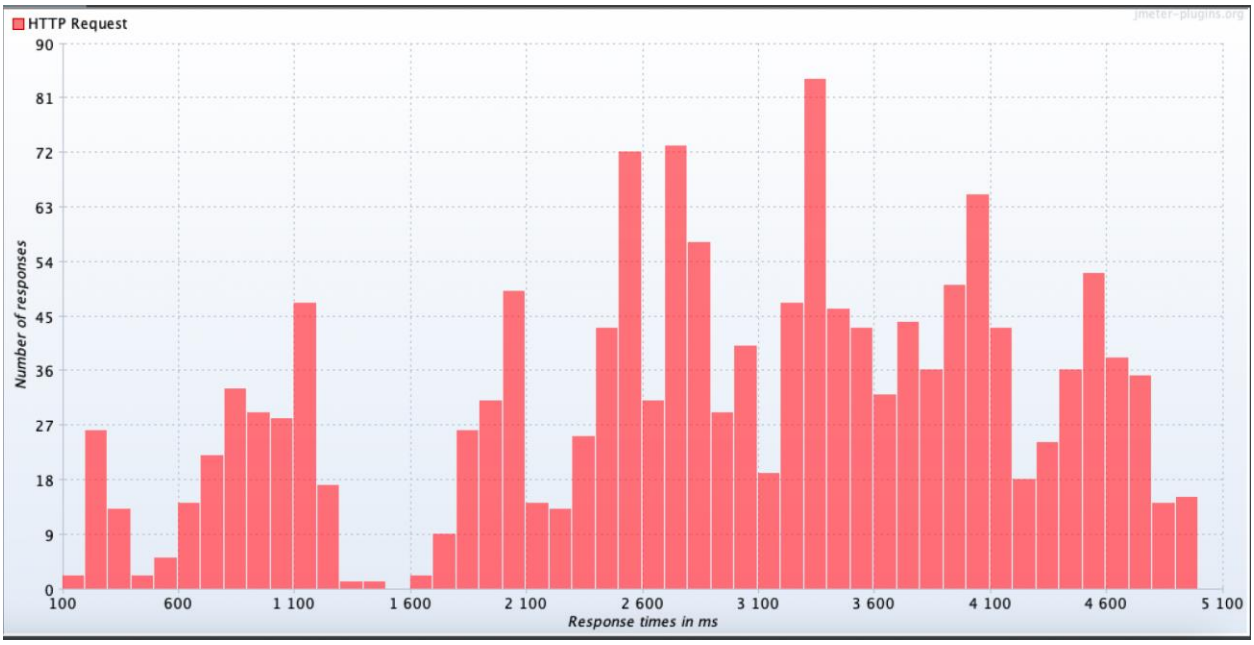
Response Codes Per Second



Response Latencies Over Time



Response Times Distribution



WHEN DELETING 3 PODS:

Deleted 1 pod while processes were requested in background.

Max Number of users: 1500

Error rate: 55.27%

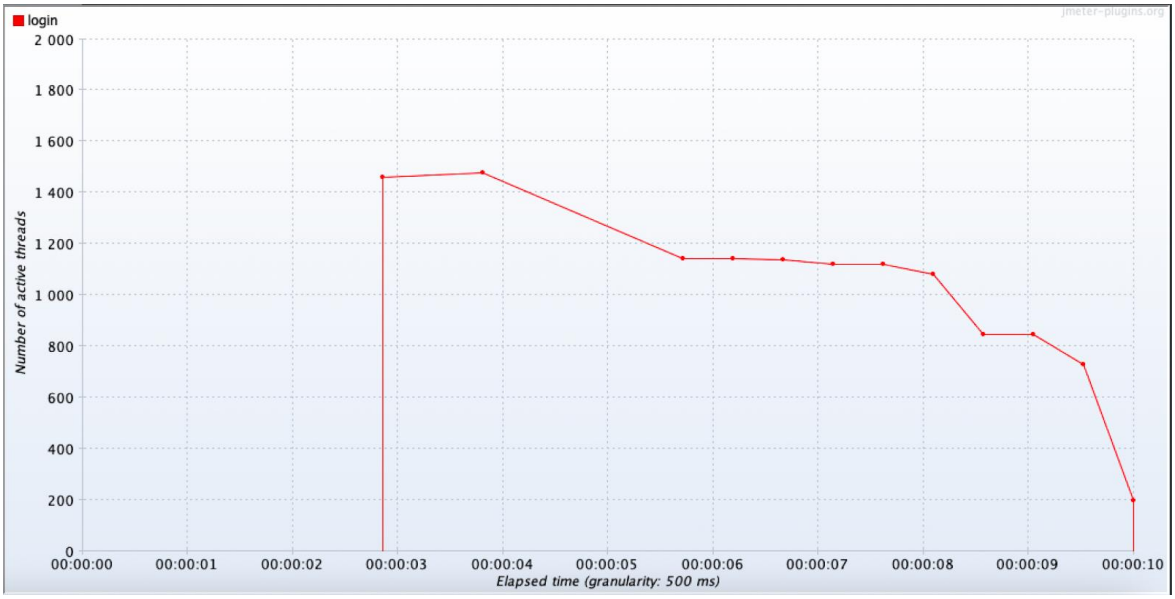
Throughput: 173.0/sec

```
(base) tanukansal@Tanus-MacBook-Pro Kub_Plotting_Service % kubectl get pods
NAME                                          READY   STATUS    RESTARTS      AGE
mongodb-deployment-97ff84f5b-ljrx7            1/1     Running   1 (19h ago)   2d16h
registry-service-deployment-858b6cc58f-6tw8j  1/1     Running   0             8s
registry-service-deployment-858b6cc58f-9fprk  1/1     Running   0             9s
registry-service-deployment-858b6cc58f-s7wxh  1/1     Running   0             42s
registry-service-deployment-858b6cc58f-t648m  1/1     Running   0             42m
registry-service-deployment-858b6cc58f-tsccm  1/1     Running   0             11h
(base) tanukansal@Tanus-MacBook-Pro Kub_Plotting_Service % kubectl delete pod registry-service-deployment-858b6cc58f-6tw8j; kubectl delete pod registry-service-deployment-858b6cc58f-9fprk;kubectl delete p
od registry-service-deployment-858b6cc58f-s7wxh;
pod "registry-service-deployment-858b6cc58f-6tw8j" deleted
pod "registry-service-deployment-858b6cc58f-9fprk" deleted
pod "registry-service-deployment-858b6cc58f-s7wxh" deleted
(base) tanukansal@Tanus-MacBook-Pro Kub_Plotting_Service %
```

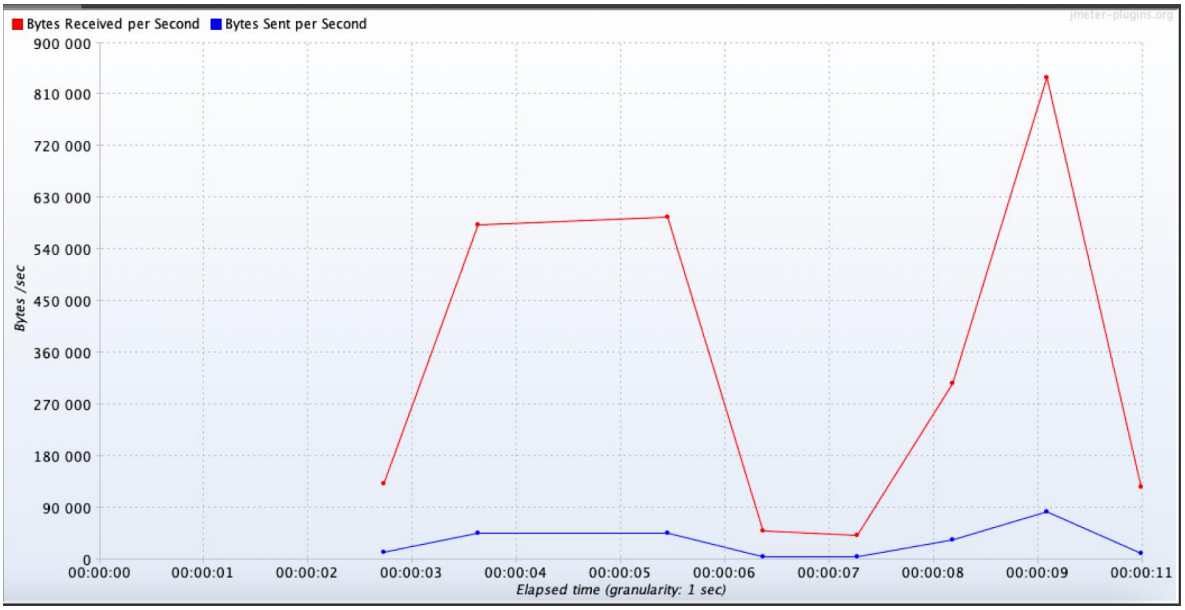
SUMMARY REPORT:

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received K...	Sent KB/sec
HTTP Requ...	1500	5059	6175	7668	8151	8440	518	8661	55.27%	173.0/sec	306.70	21.08
TOTAL	1500	5059	6175	7668	8151	8440	518	8661	55.27%	173.0/sec	306.70	21.08

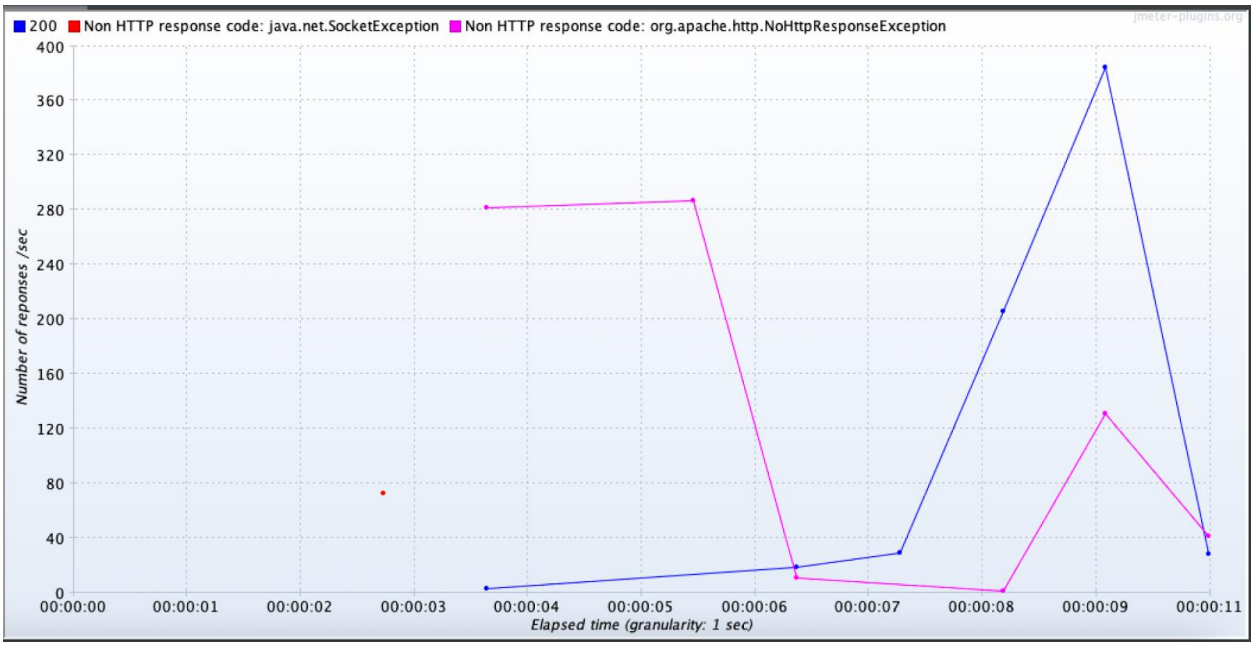
Active Threads



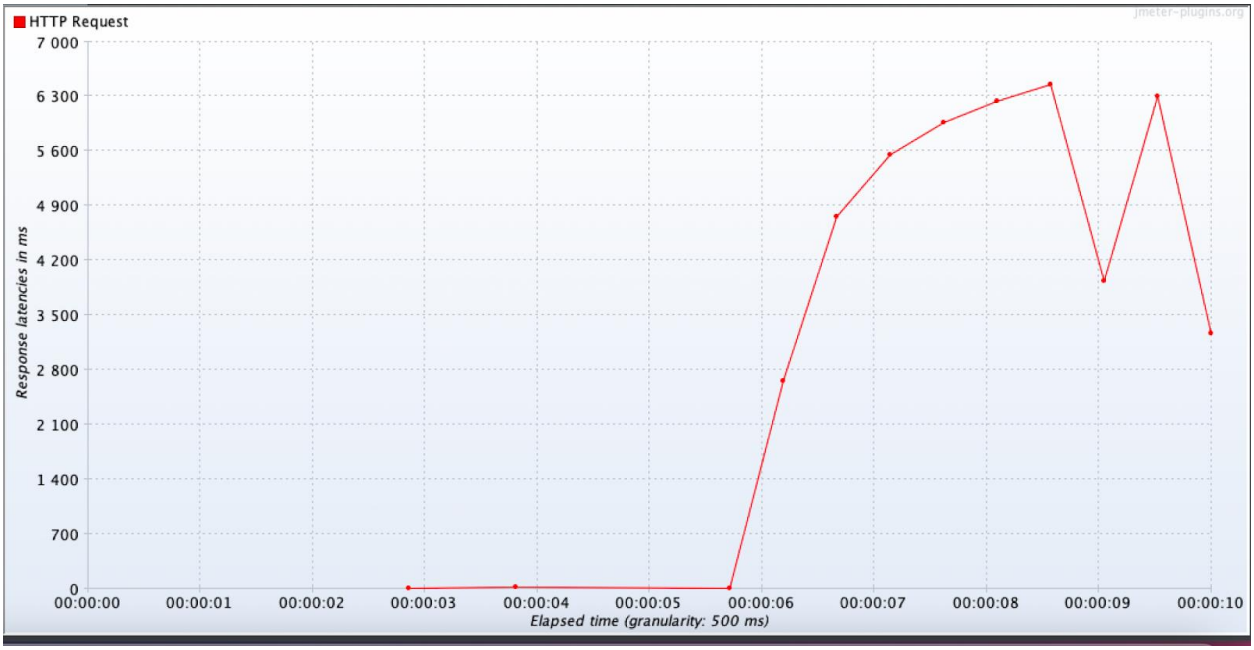
Bytes Throughput



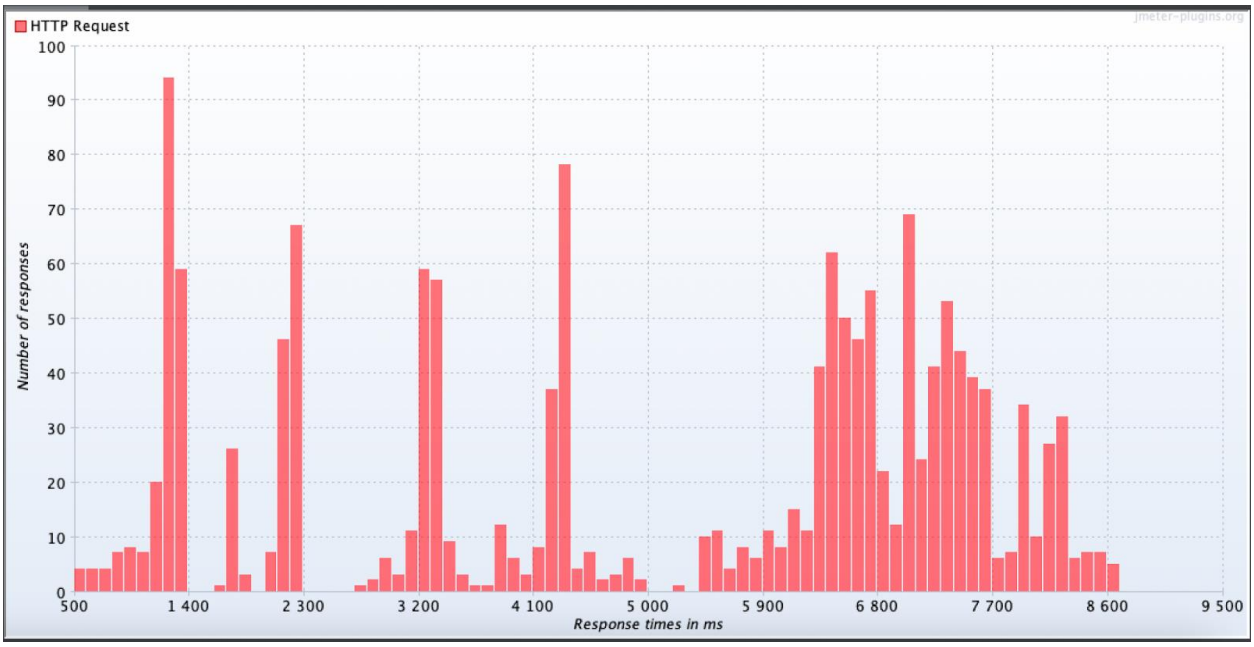
Response Codes Per Second



Response Latencies Over Time



Response Times Distribution



From the result, we concluded that the capacity of the system to handle the number of users at a time decreases. With a sudden failure the system will not crash completely, but the number of users handled at a time will decrease which results in high error rate.

While using kubernete we observed that, due to high load services(pods) failed but were automatically restarted by the kubernete.

CONCLUSIONS FROM THE GRAPH:

The number of active users depends on the processing time taken by the service. When system takes more time to complete the operation and send the response back to user, the number of active threads will increase. Once the service starts sending the response, the active users will decrease and will become zero when all responses are received. This can be observed in the active threads graph. As the plotting service taking time to process request, we can see first increase and later decrease in the active number of users in the graph, unlike the other systems, where with the time the number of users are consistently decreasing. These changes can be observed in the active threads graph.

With more number of request to the service, the processing speed of the service decreases. This results in the more latency and increases the response time. With less number of user services process each incoming request faster. This can be observed in the response latencies time graph.

If we observe the response time distribution then, in most cases it shows normal distribution pattern. We can illustrate this by the relation between number of user processing speed. Services starts with less number of request, at this time the processing is faster and the responses sent are faster as well. As the active users on the system increases the response time increases as well. At the end, in the testing, meter starts seeing requests, (suppose testing is for 800 users) and waits from the response. In this case with time the active user on the system will decrease and this results in the reduction in response time.

In conclusion, after hitting the microservice endpoints with different methods, we are observing different behavior of the application for every method. We notice that using Kubernetes replicas is quite useful to improve the performance of the system as a whole, even though it may be coming at the cost of increasing memory and CPU usage. Throughput values appear to be not hugely different when we compare with and without Kubernetes and that's because of higher processing for the high number of users for Kubernetes. But when we compare the throughput values for plotting microservice with and without Kubernetes, due to similar number of users, we see better throughput values for higher number of replicas. The tradeoff with memory and CPU usage can be bypassed by using JetStream which will host our application and improve the performance of our system.